1998-06-01

Implementation of a multiple robot frontier-based exploration system as a testbed for battlefield reconnaissance support

Hillmeyer, Patrick A.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

IMPLEMENTATION OF A MULTIPLE ROBOT FRONTIER-BASED EXPLORATION SYSTEM AS A TESTBED FOR BATTLEFIELD RECONNAISSANCE SUPPORT

by

Patrick A. Hillmeyer

June 1998

Thesis Advisor: Xiaoping Yun

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE June 1998	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE IMPLEMENTATION OF A MULTIPLE ROBOT FRONTIER-BASED EXPLORATION SYSTEM AS A TESTBED FOR BATTLEFIELD RECONNAISSANCE SUPPORT	5. FUNDING NUMBERS
6. AUTHOR(S) Hillmeyer, Patrick A.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Space and Naval Warfare Systems Center – San Diego San Diego, CA 92152-5001	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Future military battlefields will see smaller forces responsible for ever increasing geographical areas. In addition, future conflicts will occur more often in urban or built-up areas. Both of these trends argue for some type of augmentation for initial reconnaissance, continued observation, and control of lines of communication and other key terrain features. Multisensor systems, mounted on a variety of robotic platforms, can provide this type of battlefield support where it is needed most. However, before costly decisions concerning the details of such systems can be made, basic research needs to be conducted regarding their most effective composition and utilization.

Prior to this time all multiple robot studies at this institution had only taken place in simulated environments. This thesis implements a real-world multiple robot system that uses a technique known as frontier-based exploration to explore and map a laboratory or office environment. In doing so, many previously hidden aspects of multiple robot systems, unnoticeable in simulation-only studies, become evident. The results developed here are compared to results obtained elsewhere involving other robotic platforms. This research lays the foundation for future research involving multiple robots interacting as a system in a real-world environment and acting towards a common or shared goal.

14. SUBJECT TERMS Robotics, Multiple Robots, Sensor Fusion, Battlefield Reconnaissance	15. NUMBER OF PAGES 331
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

IMPLEMENTATION OF A MULTIPLE ROBOT FRONTIER-BASED EXPLORATION SYSTEM AS A TESTBED FOR BATTLEFIELD RECONNAISSANCE SUPPORT

Patrick A. Hillmeyer
Captain, United States Marine Corps
B.S., University of Minnesota, 1990

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 1998

ABSTRACT

Future military battlefields will see smaller forces responsible for ever increasing geographical areas. In addition, future conflicts will occur more often in urban or built-up areas. Both of these trends argue for some type of augmentation for initial reconnaissance, continued observation, and control of lines of communication and other key terrain features. Multisensor systems, mounted on a variety of robotic platforms, can provide this type of battlefield support where it is needed most. However, before costly decisions concerning the details of such systems can be made, basic research needs to be conducted regarding their most effective composition and utilization.

Prior to this time all multiple robot studies at this institution had only taken place in simulated environments. This thesis implements a real-world multiple robot system that uses a technique known as frontier-based exploration to explore and map a laboratory or office environment. In doing so, many previously hidden aspects of multiple robot systems, unnoticeable in simulation-only studies, become evident. The results developed here are compared to results obtained elsewhere involving other robotic platforms. This research lays the foundation for future research involving multiple robots interacting as a system in a real-world environment and acting towards a common or shared goal.

TABLE OF CONTENTS

I would [illegible]
[illegible] Program [illegible]
funding [illegible]

I would also like to [illegible]
research [illegible]
[illegible]

[illegible]
Professor [illegible]
subject material. [illegible]
rewarding.

[illegible]
of the Pattern Research Lab [illegible]
Intelligence [illegible]
insightful and thought provoking.

I would also like to thank [illegible]
assistance. Dr. [illegible]
exploration and [illegible]
multiple [illegible]
bring [illegible]
cannot [illegible]
sometimes [illegible]
learning [illegible]

Finally, I am [illegible]
parents [illegible]
then you [illegible]
support [illegible]
encouragement. I thank them [illegible]

I. INTRODUCTION

A. GENERAL

As the general downsizing of the military continues, the trend on future battlefields will be toward smaller units being responsible for scouting and securing larger areas. It is also predicted that, in the near future, 70 percent of the world's population will be in urban areas [Ref. 1]. As more and more of the general population moves into cities, future battlefields are more likely to be in urban or built-up areas. As these trends continue the need for some type of robotic support and augmentation for the small unit or individual on the ground will become greater.

Urban and built-up areas present some of the greatest challenges for military units in the areas of initial reconnaissance, continued observation, and control of lines of communication and other key terrain features. Multisensor systems, mounted on a variety of robotic platforms, can provide this type of battlefield support in areas where it is needed most. However, before making very costly decisions about the make-up of these systems it is imperative to conduct some basic research about the types of systems that are most cost effective and most efficient. This will allow the system designers to make intelligent decisions about the type and composition of systems that will be most useful on tomorrow's battlefield.

B. PROBLEM STATEMENT

As a reconnaissance system is designed there are several fundamental questions that must be asked and answered. In some missions, a large number of the simplest possible systems may be the right answer. This might be the case if all that is desired is simple detection of "something" with no detail other than the fact that there is "something" in the vicinity of the systems sensors. For other missions the best solution may be a smaller number of systems incorporating higher capability sensors, increased processing capability, improved communications resources, and greater mobility. This would be the case if the system were required to perform more complex tasks such as target identification, target tracking, or even target attack. Or perhaps the best system for the mission lies somewhere in-between these two extremes or is even a combination of them both. [Ref. 2]

This thesis will explore a comparison between the first and second options. A robotic mapping system was originally developed for a small number of very expensive, but very sensor capable, NOMAD 200 robots. By taking this same system and implementing it on a larger number of much less expensive, but less capable, NOMAD SCOUT robots the beginnings of a comparison between the two options will be possible.

In addition, many of the challenges and questions inherent to the development of a multiple robot mapping system are also present in any research involving multiple robots attempting to accomplish a common task. The problems of communication, coordination, and control apply similarly to multiple robot mine clearing systems and robotic weapon

platforms. It is hoped that the development of a multiple robot testbed can set the stage for further research in these areas at this institution.

C. OUTLINE OF THE THESIS

Chapter II provides a description of the platforms and sensors that were used for the research in this thesis, as well as the platforms used in previous work for comparison purposes. Chapter III discusses the other hardware and software common to this and other research that was used to provide connectivity and control within the systems. Chapter IV provides background on the techniques of robotic map building. Chapter V describes the exploration strategies and techniques used in this and other studies. Chapter VI describes the methods of integrating the work of multiple robots in a cooperative map making effort. Chapter VII presents the results that were obtained at the Naval Postgraduate School (NPS) based upon the system originally designed at the Naval Research Laboratory (NRL). Finally, Chapter VIII discusses the conclusions and recommendations for follow-on studies.

II. PLATFORMS AND SENSORS

This chapter provides background information so that the reader has an understanding of the NOMAD 200 and NOMAD SCOUT mobile robots and the similarities and differences between these two platforms. It will also describe the sensors available on each of the platforms, as well as some details concerning previous work done with the NOMAD 200 at both NPS and NRL. Figure 1 provides a relative size and shape comparison of the NOMAD 200 and NOMAD SCOUT.

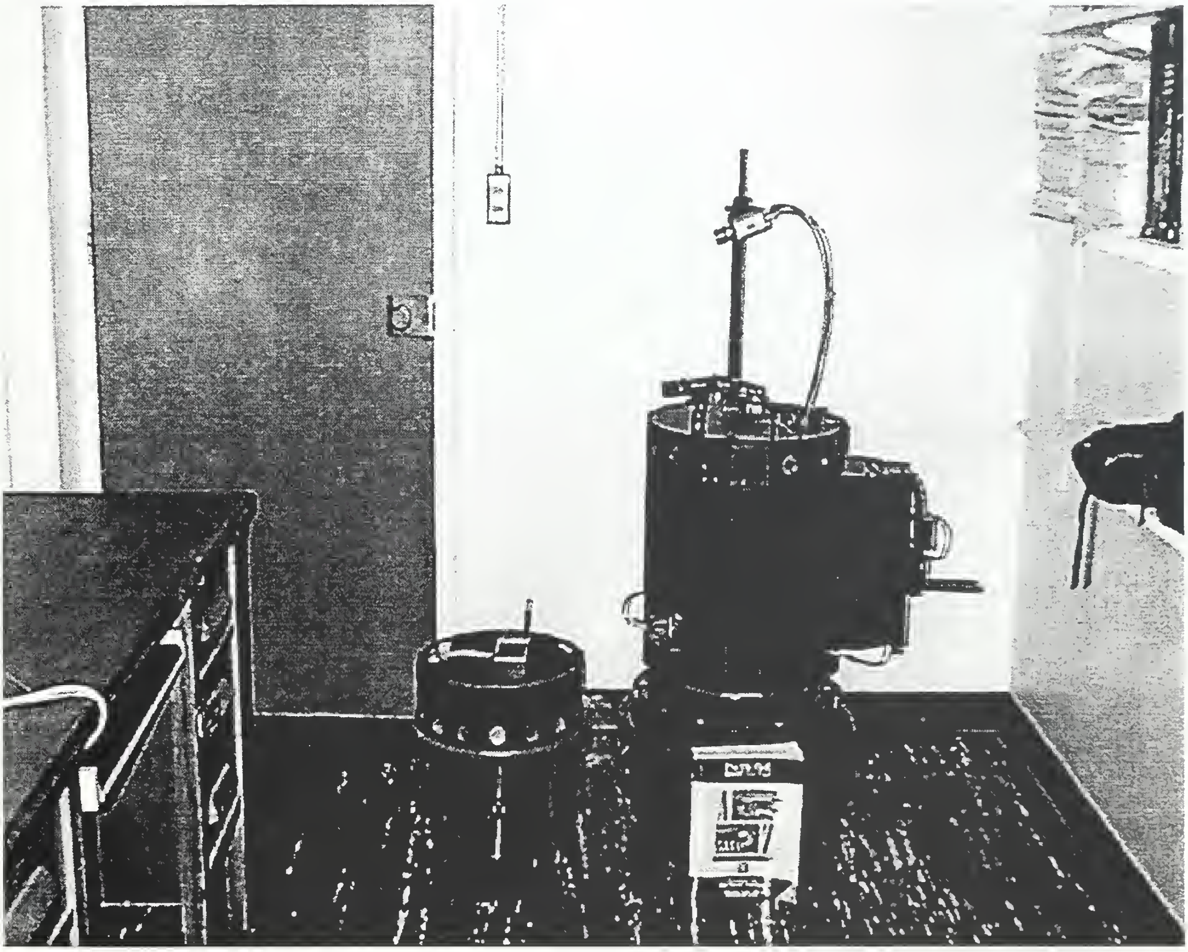


Figure 1. Relative size and shape comparison of the NOMAD SCOUT (left) and NOMAD 200 (right) – note telephone book in front of NOMAD 200 for reference.

A. NOMAD 200 MOBILE ROBOT

The NOMAD 200 is an integrated mobile robot system with four sensory modules including tactile, infrared, sonar and laser sensors. There are multiple onboard computers that provide sensor and motor control, as well as providing communication to the host computer via a wireless Ethernet system [Ref. 3]. Figure 2 provides a detailed picture of the NOMAD 200.

1. Mechanical Description

The NOMAD 200 base chassis is driven by a three-wheel synchronous drive mechanism, using one motor to drive all of the wheels and a second motor to steer all of the wheels. The robot has a zero gyro-radius, meaning that it can rotate about its own center. It can translate at up to 24 inches per second and rotate at a maximum rate of 60° per second. The base is 18 inches in diameter, which extends to 21 inches with the bumper installed. The NOMAD 200 stands 31 inches tall, excluding any additional sensors added on top of the robot [Ref. 4]. The turret (which all the sensor systems are mounted on) can be rotated independently of the base. [Ref. 4]

2. Sensor Systems

The basic NOMAD 200 sensor array includes tactile (bumper) sensors, infrared sensors, sonar sensors, and laser sensors. In addition to these sensor systems, the robot also has an odometric system that tracks the robot's movements. The encoder resolution

on this odometric system is 18 counts/cm for translation and 1510 counts/degree for robot steering and movement of the turret.

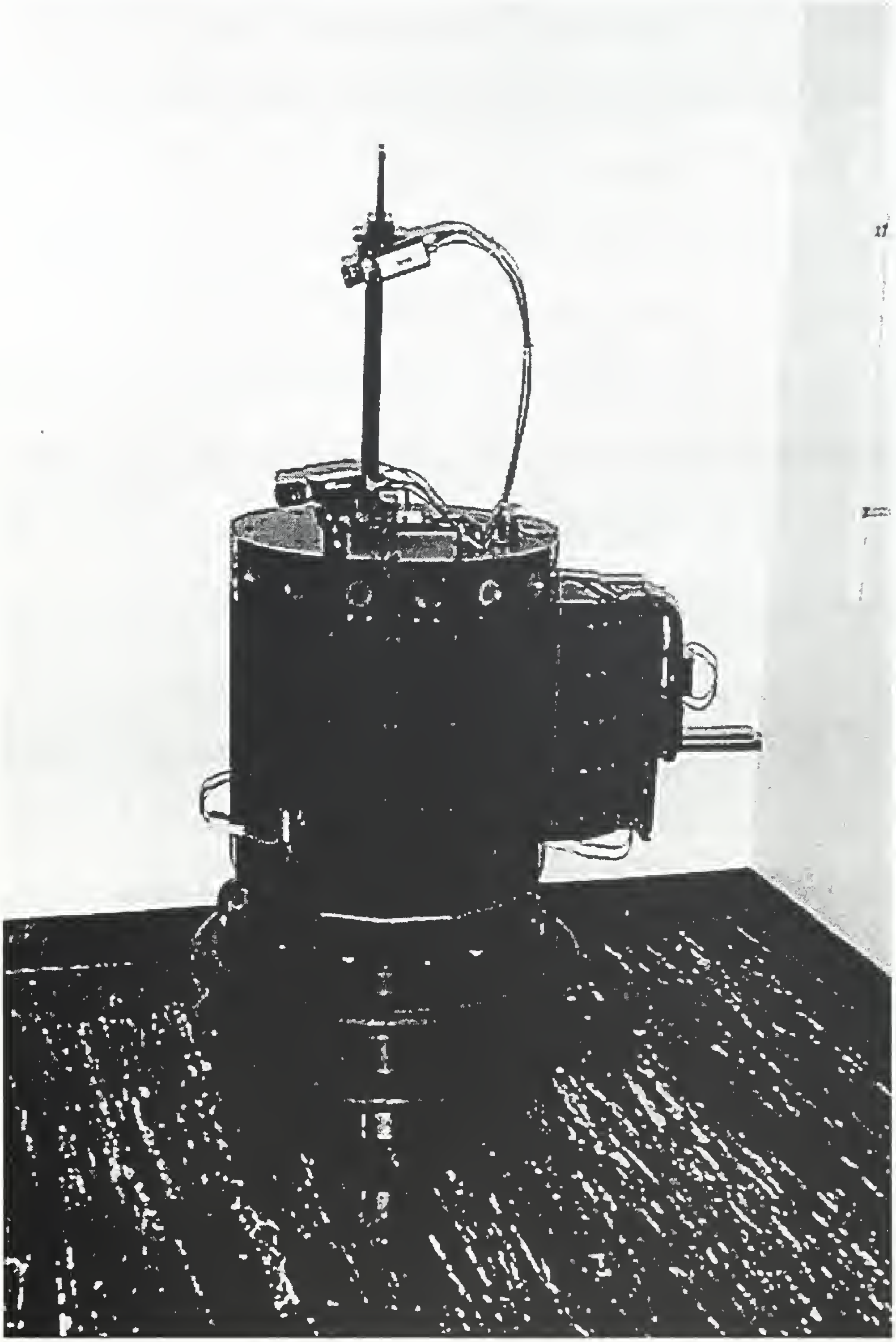


Figure 2. Detailed close-up picture of the NOMAD 200.

The tactile system consists of a bumper ring positioned over 20 independent pressure-sensitive sensors. These simple on-off sensors are interleaved around the bumper ring in order to provide 360° coverage with 18° resolution. [Ref. 4]

The infrared sensors aboard the NOMAD 200 incorporate a 16 channel, reflective intensity based, infrared ranging system that provides 360° of coverage. Each of the 16 sensors is composed of two light emitting diode (LED) emitters and a photodiode detector enclosed in a delrin housing. The range from the sensor to the object(s) is determined by the amount of light from the emitters that is reflected back to the detectors after striking the object(s). The reflectivity of the object greatly affects the reading. Thus the system needs to be calibrated for the environment in which it is to be used. With appropriate calibration, range accuracy is within 5% from 0 to 24 inches from the sensor. [Ref. 5]

The sonar sensors on the robot are composed of a 16 channel, time-of-flight based, sonar ranging system. The system uses standard Polaroid transducers. Each transducer has a beam width of 25°. Range from the sensor to object(s) is determined by the time of flight of the acoustic signal generated by the transducer and reflected back by the object(s). The user has the option to control the firing sequence of the 16 individual sonar sensors mounted about the circumference of the robot. To minimize the potential for sensor interaction, a non-sequential firing sequence is recommended. [Ref. 6]

The laser sensor on the NOMAD 200 is a two-dimensional, triangulation-based laser ranging system. A laser diode is used as the light source and a charged-coupled device (CCD) array camera is used to generate an image. The laser diode produces a

horizontal “plane” of light. The CCD camera is placed vertically above this “plane” and inclined downward. Any object intersecting this plane forms a light stripe on the image generated by the CCD camera. The range to this object is found by determining the position of this light stripe along the scan lines of the camera. This system has an operating range from 12 to 120 inches. [Ref. 7]

3. Previous Work with this Platform

The NOMAD 200 has been and continues to be a very popular research platform at NPS and elsewhere. There exists an extensive body of work involving the NOMAD 200 in several areas in the field of robotics. Some of the more recent work at NPS has involved localization of the robot position in an unknown environment [Ref. 8, 9] and geometric formation and movement in formation of multiple robots in simulation [Ref. 3]. Because NPS only has a single NOMAD 200 (due primarily to the expense of the platform), all work involving multiple robots had to be simulated until very recently. This single robot limitation coupled with the high logistics cost of setting up a very complicated robot platform have been major factors in limiting research performed with real, vice simulated, robots at NPS.

NRL has also done extensive research using the NOMAD 200. Their acquisition of two NOMAD 200 robots has allowed them to conduct more actual research involving multiple robots in addition to simulations. The basis for this thesis is an adaptation of some of their work (described below) in order to form a testbed for actual multiple robot work here at NPS involving less costly platforms.

B. NOMAD SCOUT MOBILE ROBOT

The NOMAD SCOUT is an integrated mobile robot system with ultrasonic and tactile sensors, as well as an odometric system. It uses a multiprocessor, low-level control system that controls the sensing, motion, and communications. At a high level, the SCOUT is controlled either by a laptop, mounted on top, or a remote workstation communicating via radio modem. The SCOUT is code compatible with NOMAD 200 class robots [Ref. 10], which was a very important consideration in its choice as a research platform at NPS. Figure 3 provides a detailed picture of the NOMAD SCOUT.

1. Mechanical Description

The NOMAD SCOUT is a two-degree of freedom (DOF) differential drive robot. The drive is set about the geometric center of the robot, which allows the robot to turn or rotate about its own axis. The NOMAD SCOUT has a maximum speed of one meter per second with a maximum acceleration of two meters per second squared. The robot is .38 meters in diameter and is .34 meters in height. Without batteries, the unit weighs 23 kilograms. [Ref. 11]

2. Sensor Systems

The basic NOMAD SCOUT sensor array includes tactile (bumper) sensors and sonar sensors. In addition to these sensor systems, the NOMAD SCOUT also has an odometric system that tracks the robot's movements. The encoder resolution on this

odometric system is 167 counts/cm for translation and 45 counts/degree for robot rotation.

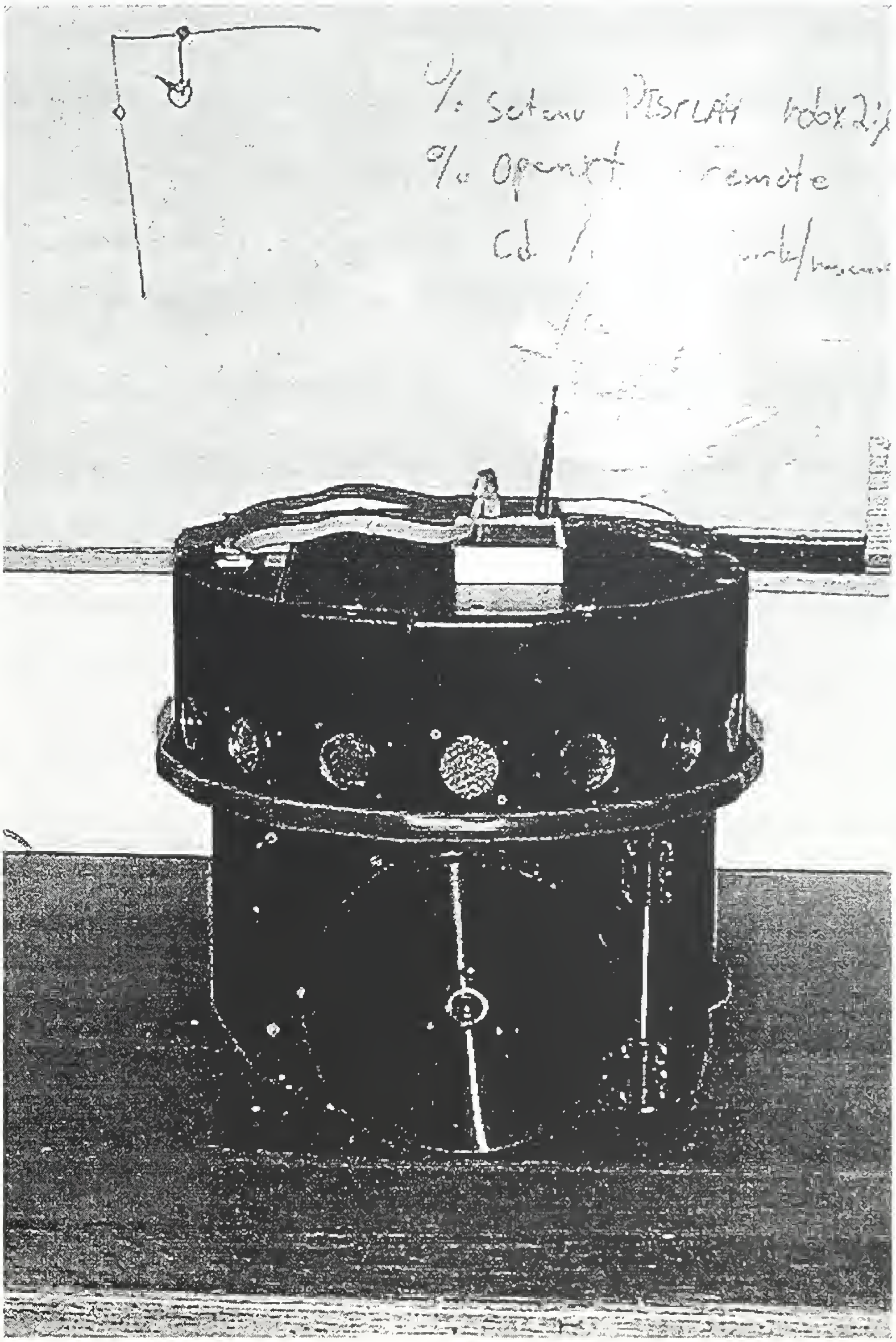


Figure 3. Detailed close-up picture of the NOMAD SCOUT with radio modem.

The tactile system uses a ribbon switch enclosed in an energy absorbing neoprene channel to provide 360-degree coverage. The ultrasonic system uses 16 independent sonar sensors. The effective range of the ultrasonic sensors is 6 to 255 inches. This sonar sensor is basically identical to the one installed in the NOMAD 200 with slight differences due to the smaller diameter of the NOMAD SCOUT. [Ref. 11]

C. COMMUNICATION AND COMPUTATIONAL RESOURCES

No description of the platform would be complete without mention of the software and hardware that allow the robots to operate. A thorough knowledge of the underlying hardware and software is essential in understanding the model being used in this research.

1. Software

The Nomadic Host Development Environment (NHDE) is a full-featured, object-based mobile robot software development package for both the NOMAD 200 and NOMAD SCOUT mobile robots [Ref. 12]. The complete package provides the control and graphics interfaces as well as a very realistic simulation tool for program testing. By using the supplied development package it is much easier to concentrate research on higher level issues of motion planning and control because most of the lower level issues of sensor and motor control are handled by the included software.

The control interface allows for programming the NOMAD 200 or NOMAD SCOUT using a high-level programming language (either C, C++ or Lisp) and linking to a supplied library [Ref. 12]. Built-in to the supplied library are interfaces to the supplied driver software that handle lower level functions such as sensor and motor control. This allows for a higher level of abstraction in the researcher's approach. The graphical user interface and simulator are accessible when the user runs the executable server program, *Nserver* [Ref. 3].

The graphical user interface in the NHDE is based on the OSF/Motif graphics toolkit for the X Window System [Ref. 12]. The graphical interface can display information on up to six robots simultaneously. There are several different windows displayed in the complete graphical user interface. First, there is the world (map) window which gives an overall view of the environment (real or simulated) that the robots are in, as well as the positions of the robots relative to the environment and one another. Secondly, there is a robot window, with one copy per robot, which contains information about each individual robot. This information includes the current command being executed, position and orientation information, and sensor information. Along with each robot window, there are two more windows that give more detailed information about current sensor readings. These two windows are usually used to display a graphical representation of the sonar and infrared returns of each robot's sensors. Detailed information about each robot can be saved as a setup file (*robot.setup*) [Ref. 3]. This includes information about the model of robot (NOMAD 200 or NOMAD SCOUT) being used.

The Nomadic Simulator is a fully functional mobile robot simulator that can accurately model most environments, the robot's motion, and its sensing capabilities. There is a high degree of correlation between the simulated world and the real world. If a program will not run on the simulator it definitely will not work on a real robot. The simulation tool allows the researcher to build a controlled environment in which to develop and debug programs. Using a graphical drawing tool a user can draw a map in the world window to simulate the desired surroundings. This file can be saved as a setup file for the world (*world.setup*). In addition, once a program is running properly on a simulated robot, it can be switched to a real robot via a pull-down menu within the simulator. Once this is done the graphical interface will then begin to display information from the real robot vice the simulated one while commands from the program will control the real robot via the server program (*Nserver*). [Ref. 3]

The NHDE also incorporates several other very convenient features that aid the researcher. There is a record and playback tool that that allows for sensor data and/or executed commands to be stored for later analysis, as well as providing for an instant replay capability. This is an invaluable debugging tool. There is also a console available on *Nserver* that allows the user to directly input to the robot any possible command. This is a very handy option for checking the robot's sensors or making small, subtle adjustments in the robots location during experiments. In addition, there is an on-screen, software joystick that allows the user to remotely drive the robot. This is often used to move actual robots around in the real world while simultaneously collecting sensor data.

This allows the researcher to write software that collects and manipulates sensor data without having to also write code to handle the robot's motion.

Figure 4 shows a typical NHDE display from an experiment involving two robots in a simulated environment. Also shown in the figure are many of the features described above such as the global map window, the robot window with its associated sensor windows, the record and playback window, the command console window, and the software joystick window. Also demonstrated in the robot window is another feature available in NHDE. The user has the option to display raw sensor return data or "hits" in the robot window as well as a copy of the global map. This provides a quick visual reference to the researcher indicating whether or not the robot's sensors are functioning properly.

Using *Nserver* in conjunction with the graphical interface and simulation tool is a very convenient way to test and debug software in simulation for subsequent use on a real robot. However, because of the client-server architecture, testing client programs on the real robot via *Nserver* may slow the control and data return rates because *Nserver* acts as a router. Once a program is working properly it can be recompiled to use the control interface library directly without the need for *Nserver* to be running concurrently. This is a very simple process because of the efficient design of the NHDE software.

Each of the application programs for each robot, as well as the *Nserver* when used, can run simultaneously as separate processes under the UNIX operating system. All communications between the host processes that are controlling the robots are

handled as communications between UNIX processes using the TCP/IP protocols and a server-client architecture. This will be described in more detail in Chapter IV.

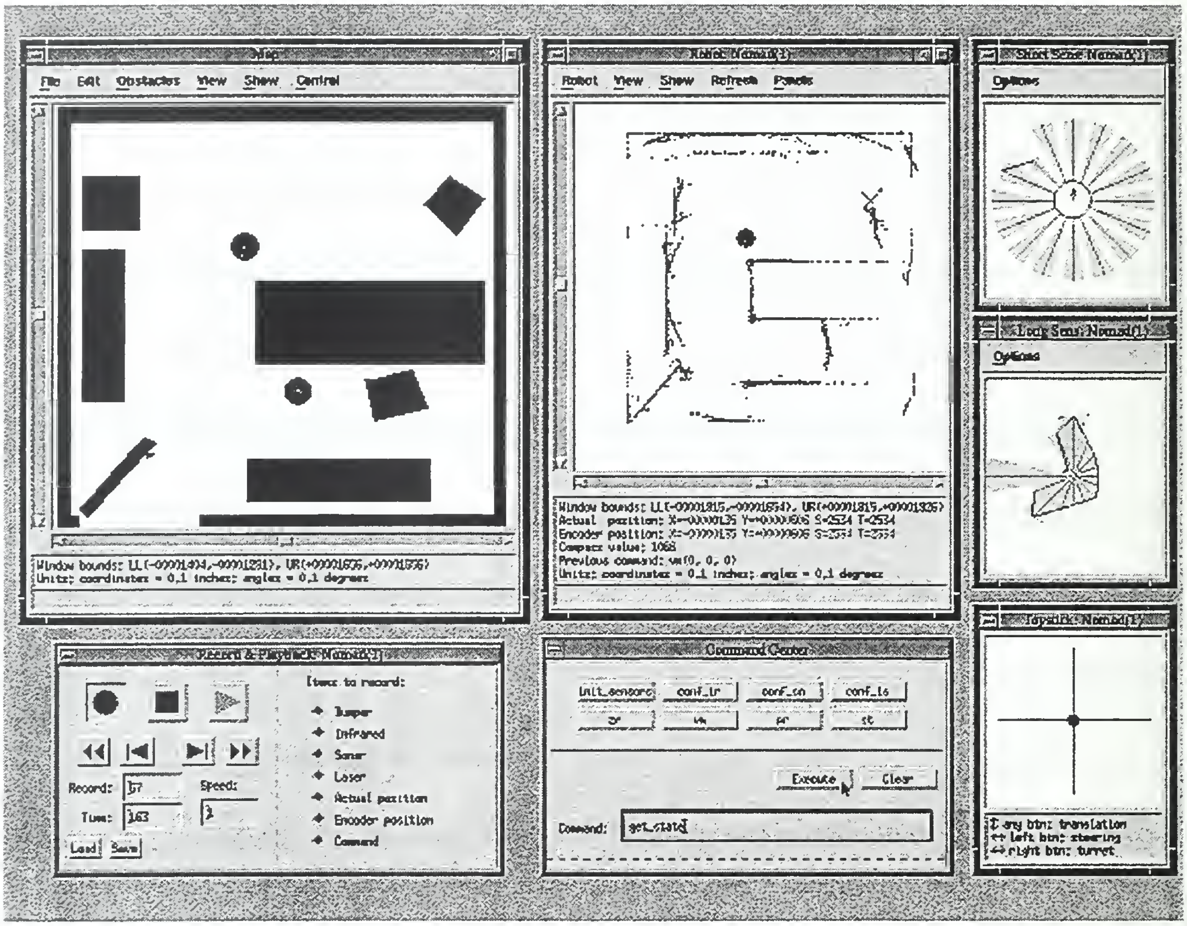


Figure 4. Typical graphical display. (From Ref. [12])

2. Hardware

As mentioned above, each program runs as a separate UNIX process. Whenever practical each process is run on a separate Sun workstation because the frontier-based exploration program is computationally demanding. In addition, running each robot with a separate workstation is more faithful to the system that is being modeled as described in Chapter IV.

Communications from host process to host process are handled as described above and in more detail in Chapter IV. Communications between the host process and the robot it controls are handled via radio Ethernet. Each robot has a 2.4 GHz radio modem. This radio modem is assigned an IP address that the host process uses to route instructions to the robot [Ref. 13]. The radio modem connects to a wireless access point that provides connectivity between the radio modem and the rest of the network [Ref. 14].

III. EVIDENCE GRID BASED MAPPING TECHNIQUES

Over the years many methods have been developed to convert sensor data from robots into useful maps. At times it seems that there are as many different types of robotic mapping techniques as there are robots. Each group of researchers has approached the problem with a slightly different variation or procedure. However, one major method that has proven very successful is called the "grid method."

A. OVERVIEW OF GEOMETRIC MAPPING TECHNIQUES

A geometric map represents objects according to their geometric relationships. It can be a grid map, or a more abstracted map, such as a line map or a polygon map [Ref. 15]. A geometric map also has the advantage of being easily interpreted by humans trying to match the map with the area that it represents. The key is finding the method that builds the best map.

One of the problems with building a map using simple lines and polygons is that the mapping capability breaks down quickly in a non-simulated environment. These methods depend on interpreting small amounts of sensor return data and mapping points, lines, or surfaces to that data. This can work very well in a simulated environment where obstacles tend to be composed of simple geometric shapes and straight lines, but the real world is not made up of such convenient shapes. Often times such methods map false obstacles or incorrectly shaped obstacles based on extraneous or incorrect sensor data. This is especially a problem when dealing with sensors that return much "noisy" data by

their nature, as do sonar sensors. So these line and polygon mapping techniques are said to be lacking robustness.

The grid technique was developed as a way to overcome many of the problems described above. When using a grid method it is not necessary to make assumptions about the shape or size of an object being mapped. Simply plotting enough sensor return points on a grid forms a recognizable map that can be used by both robots and humans. Once enough points are plotted, edge detection techniques can then be used to pick out walls, obstacles, unexplored areas, and other terrain features. More about edge detection techniques and their uses in mapping will be discussed in Chapter IV.

B. SIMPLE PLOTTING OF SENSOR DATA

Perhaps the most basic of the grid methods is simply plotting the data returned by the robot's sensors and marking areas within the sensors' range as either occupied or unoccupied. This approach has the advantage of simplicity and can produce a reasonably good map in a well-defined simulated environment. However, in a real world environment using only sonar as a sensor this method quickly breaks down unless very tight constraints are set on the range of data used. This was the first mapping technique attempted for this research in conjunction with a single robot. Although this method was later abandoned in favor of a technique better suited for a multiple robot system, it is still useful in depicting some of the complexities of robotic mapping systems.

Figures 5 and 6 illustrate a comparison of simulated and real-world maps constructed by the simple plotting of sonar return data with varying range limitations

imposed on the displayed data. Figure 5(a) is a typical simulated environment used to test various robotic map-making techniques. The environment is a roughly 325 by 275 inch rectangle with several large geometrically shaped obstacles placed within it. Figure 6(a) is a photo of an aisle between two laboratory benches that was used to test map-making methods in a real environment. The aisle has several chairs with metal legs along the robot's projected path as well as open spaces beneath the benches. In both the simulated and real worlds the robot is remotely moved through the environment via the software joystick described above.

At the same time the user remotely moves the robot, another process is running which collects the sonar return data and the robot position and orientation (*pose*) data and writes it to a data file. Pose data is very important in converting the sonar returns for a given robot position and orientation into data that can be mapped onto a common coordinate system. After the data were collected, a MATLAB routine read the data file, transformed the sonar return and pose data, and plotted the resulting map. Along with the sonar return data the robot's path in the simulated or real world is also plotted as a dotted line in the resulting map. In this case the map was generated after maneuvering the robot, but *Nserver* also allows for the raw sensor returns to be plotted in real time within the robot window.

In Figures 5(b) and 6(b) the reliable sonar range is set to 255 inches (the maximum rated reliable range according to the manufacturer's specifications). Thus, the mapping program plots all sonar return data that is below 255 inches. A return of 255 inches is regarded as open space and not plotted. In the simulated world this produces a relatively

good map with some noise at corners and other line intersections. In the real world there is much noise and what appear to be many extraneous returns. This noise and the apparent false returns will be discussed in more detail in Chapter IV.

In Figures 5(c-e) and 6(c-e) the reliable sonar range is steadily reduced and a larger percentage of the raw sonar returns eliminated and not plotted. Correspondingly, as the outlying returns are discarded, the data that is plotted produces clearer and less noisy maps. Unfortunately, as can be seen very well in Figure 5(e), dropping the longer returns in the larger simulated environment resulted in the robot path being too distant from several obstacle walls, preventing them from being mapped. This illustrates the tradeoff between reducing the reliable sonar return range in order to get quality data, and forcing the robot to travel further in order to close in on all mappable objects in the environment. More about this tradeoff will be discussed in greater detail in Chapter V.

Plotting every sensor return does not prove to be the best method of constructing a useable map. The same simplicity that makes it so easy to implement also proves to be its downfall in real world situations. It is possible to fuse maps together with this method by converting all returns to a common coordinate system, but the main problem is that all data is given the same amount of validity. What is needed is a method to weigh, or measure, the "goodness" of sensor data from multiple sensors at multiple positions and build a map accordingly.

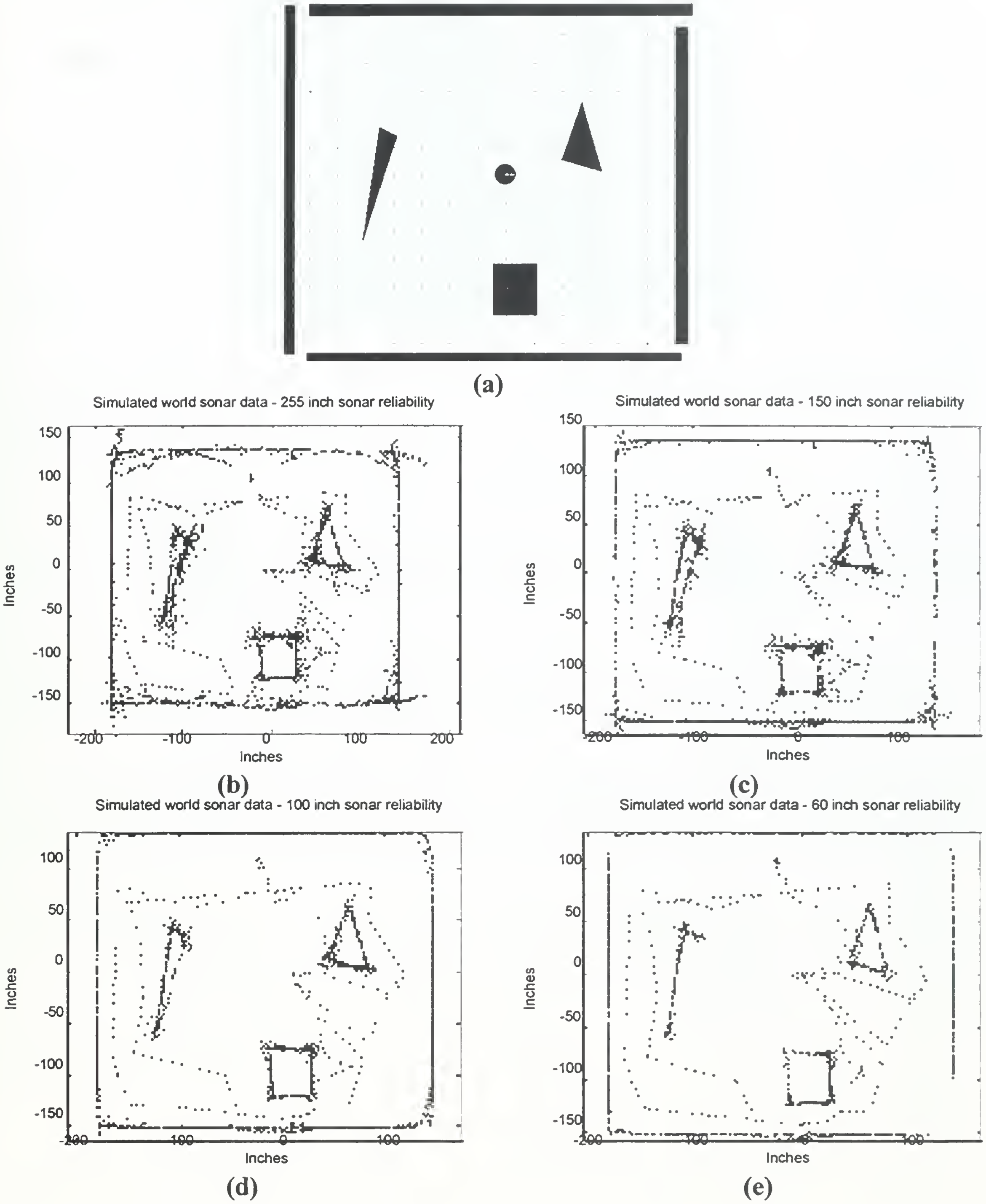


Figure 5. Illustration of simulated environment and the resulting sonar maps formed by maneuvering the simulated robot throughout it, collecting sonar range data, and plotting subsets of that range data based on estimated reliability.

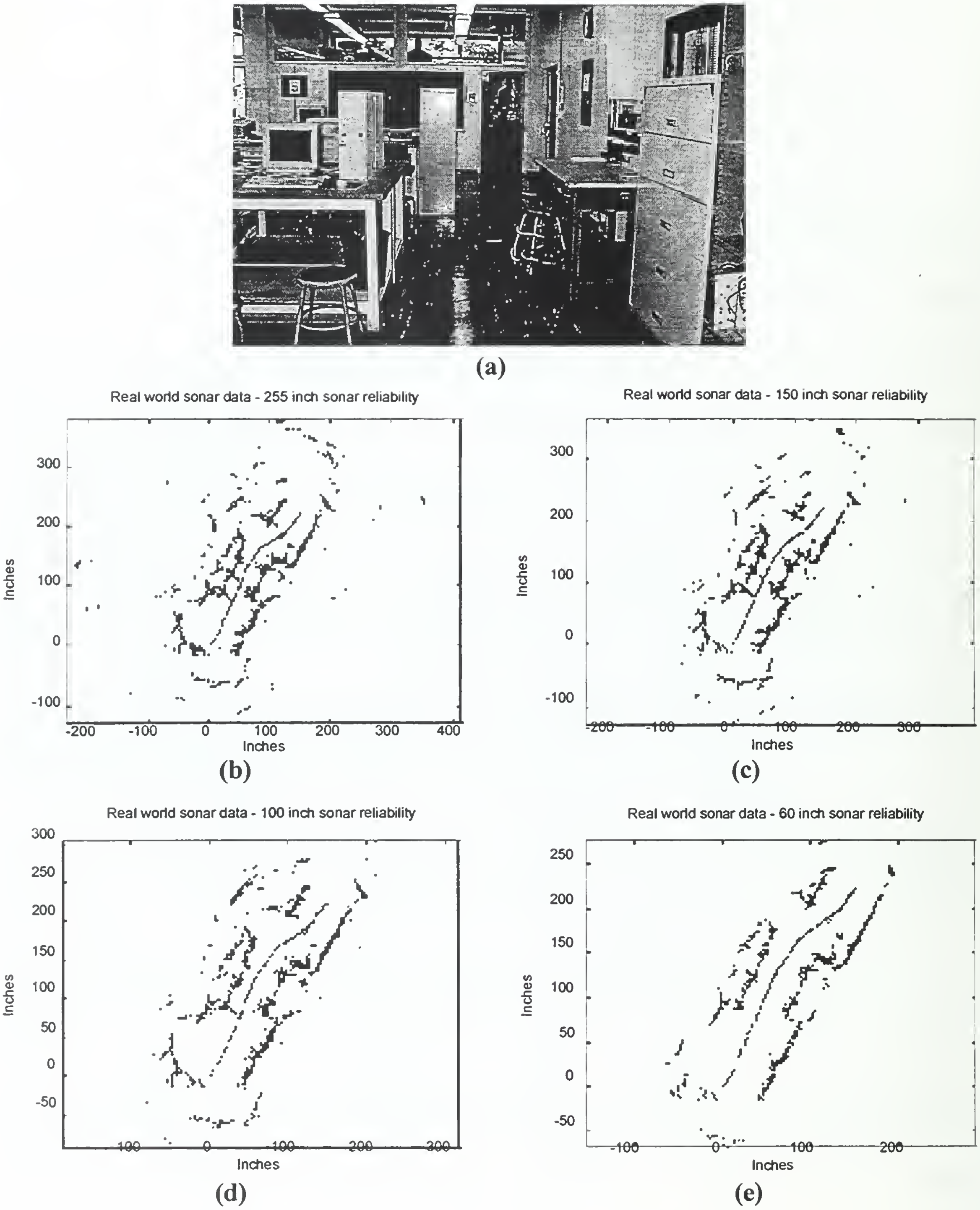


Figure 6. Illustration of real world environment and the resulting sonar maps formed by maneuvering the actual robot throughout it, collecting sonar range data, and plotting subsets of that range data based on estimated reliability.

C. THE EVIDENCE GRID METHOD

The evidence grid method was developed as a technique to create high resolution maps from wide-angle sonar. This approach allows range measurements from multiple points of view to be systematically integrated into a common map. The integration technique allows for multiple readings of an area to either reinforce or refute one another as to the whether or not the area is occupied. As more sensor data is added the definition of the map improves. [Ref. 16]

In the evidence grid method, the area to be mapped is divided into a grid with M X N cells. In each of these cells is stored a set of information regarding the best estimate as to what that cell contains. This "evidence" can be many different things, such as the surface orientation or color of whatever is in the cell, but for mapping perhaps the most useful information is occupancy information concerning the cell [Ref. 17]. Early versions of this method [Ref. 16, 18] stored this occupancy information as a two part record with a status of either *unknown*, *empty*, or *occupied* and an associated certainty factor of either 0, -1 to 0, or 0 to 1 respectively. Early versions also used ad hoc formulas to combine this information about each cell into a useable map.

Later implementations [Ref. 17, 19, 20] eliminated the two part occupancy record and replaced it with a single value representing the probability that the cell is occupied. This technique also allowed for a better method of combining and integrating sensor data using a variation of Bayes theorem. In this representation, an unknown or unexplored cell would have an occupancy probability of 0.5. As more sensor data becomes available this

probability changes accordingly. The details of how this sensor data is integrated together are the topic of the next section.

D. FUSING SENSOR DATA USING AN EVIDENCE GRID

The major advantages of the evidence grid method over previous methods are the ability to weigh or measure the "goodness" of the sensor data and the ability to fuse this data in a simple, yet effective, manner. The best way to demonstrate how the sensor data is fused is to first present a graphical representation and then go more in depth into the mathematics behind it.

1. Graphical Presentation

Figure 7 shows a sequence of images simulating the fusion of sonar sensor data from two different points. In this sequence, the circles labeled *A* and *B* represent locations at which a sonar sensor sends out a pulse and receives a return. Points *A* and *B* could be different sensors on the same robot, the same sensor on a single robot at a different time, location and/or orientation, or two different sensors on two separate robots. The advantage of this sensor fusion method is that for all practical purposes, exactly what they are does not matter. The only thing that matters is that the readings be relatively independent of one another. For the purposes of this explanation, they will be referred to as sensor *A* and sensor *B*. The background grid will represent the evidence grid that will be developed in order to map the region. As per most recent implementations of

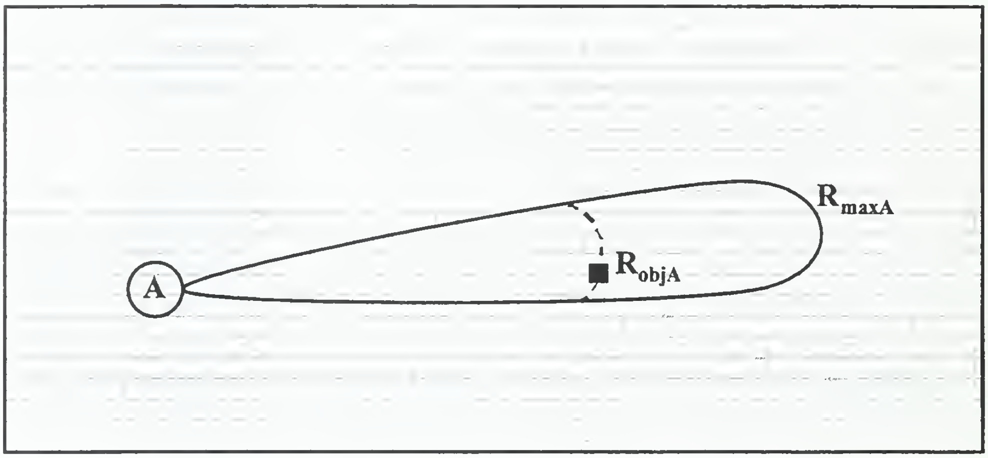
the evidence grid model it will be assumed that all the cells on the grid will be initialized to an initial occupancy probability of 0.5.

In Figure 7(a) the relative geometric locations of A and B are shown, along with some object in the distance. Figure 7(b) shows a two-dimensional “slice” of a three-dimensional sonar cone emanating from A. R_{maxA} is the maximum effective range of sensor A. R_{objA} is the range at which the object is detected some distance away from A. Because of the wide-angle nature of the sonar sensor the object is known to be only somewhere on a certain surface [Ref. 20]. In this case the dashed line represents the edge of the circular “slice” of the sonar cone along which the object might lie. So at this point it is known that an object lies somewhere within a constrained region.

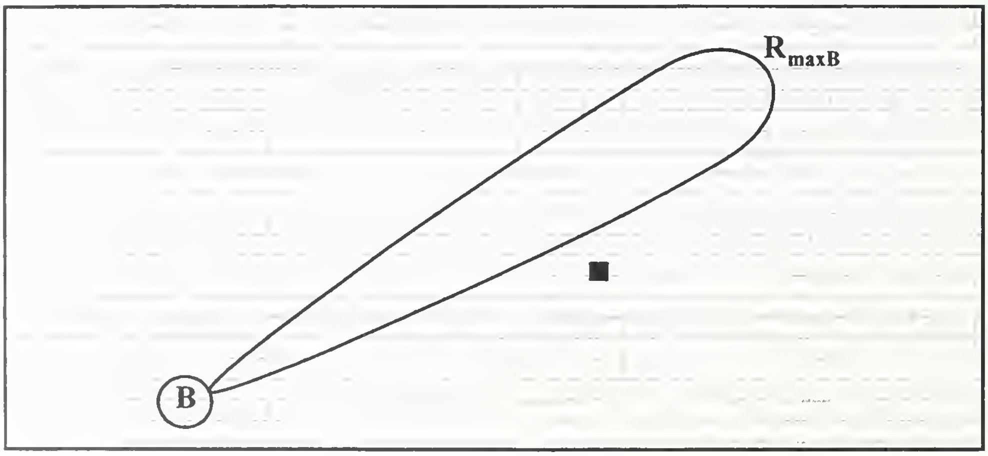
The shape of that region and its distance from A are also known, but there is not enough information at this point to make an assumption about exactly where in that region the obstacle exists. Under the evidence grid model the occupancy probabilities of the cells along that entire region would be increased. In the two-dimensional case this would mean all the cells along the dashed line. The amount of increase might vary depending on several factors such as distance from target, angle from sensor, or other measures of sensor data quality, but the main point is that the occupancy probabilities along the assumed obstacle location would be increased relative to the surrounding cells.

Figure 7(c) shows the two-dimensional representation of the sonar cone emanating from B. Again, B may be the same sensor as A at different time, location, and/or orientation or a separate sensor on the same or a different robot. In this case B does not sense the obstacle within the region it is observing.

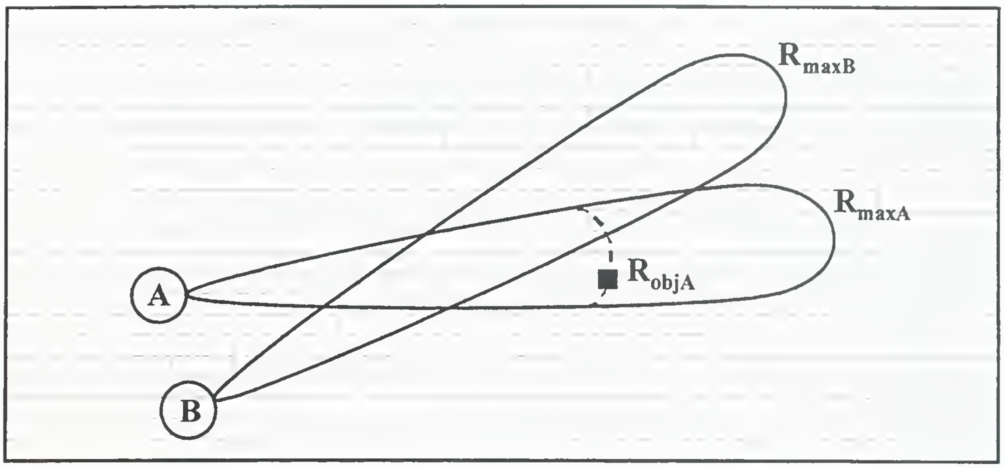
(a)



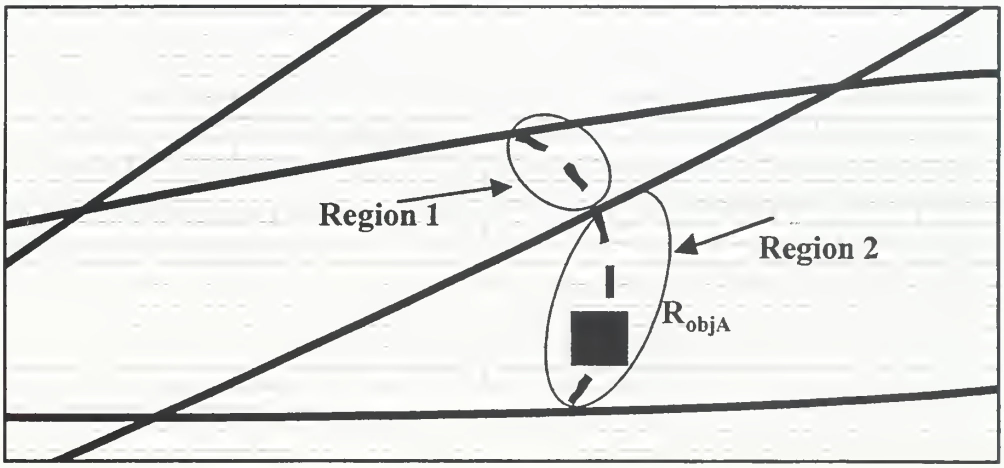
(b)



(c)



(d)



(e)

Figure 7. Example of fusion of sonar return data from two geographically different sonar sensors.

Figure 7(d) displays both the sensor readings from *A* and *B* simultaneously. In this case part of the sensor reading from *B* overlaps some of the region of the sensor reading from *A* where the cells had had their occupancy probability raised relative to the surrounding cells. However, the data from sensor *B* now provides additional information

concerning portions of this area and that information can be used to adjust the occupancy probabilities in that region accordingly.

Figure 7(e) shows an enlarged view of the area where the sensor readings from *A* and *B* intersect. In the evidence grid model the information from *A* and *B* would be combined in such a way as to lower the occupancy probabilities of the cells within the circle marked *Region 1* that are in view from both *A* and *B*. At first it would seem logical to also increase the occupancy probabilities of those cells within the circle marked *Region 2* as well. However, because the evidence grid model used considers each sensor reading to be relatively independent of every other sensor reading, the reading from *B* cannot be used to adjust the occupancy probabilities of the cells in *Region 2* because those cells are not in view of *B*.

Depending on the detailed specifics of the model chosen, the occupancy probabilities of the cells in *Region 1* might still be higher than their immediate neighbors that were always considered empty, but they would always be lower than those of the cells in *Region 2*. So even though the cells in *Region 2* are not directly changed through this process, their occupancy values are now the local maxima for the overall area within the large evidence grid shown in the figure.

Figure 7 illustrates the evidence grid method on a small scale. Now imagine it on a much larger scale with multiple sensors on multiple platforms and many hundreds to thousands of relatively independent sensor readings from a multitude of ranges and orientations. It is by combining all of these together that a map is created using the evidence grid method. The mathematical details of this process are described below.

2. Mathematical Presentation

Perhaps the clearest and most concise mathematical description of the evidence grid method of combining sensor data can be found in [Ref. 17]. What follows in this section is a condensed version of that work presented here for clarity and completeness.

Let $p(A|B)$ represent the best estimate of the likelihood of situation A given that information B has been received. A and B mean either "a certain region of space is occupied," (written o), "a certain region of space is unoccupied", (written $\bar{o}$), or they represent a sensor reading. By definition, $p(A|B) = p(A \cap B)/p(B)$. The quantity $p(A)$ represents the estimate of A given no new information. The alternative to situation A is written $\bar{A}$, (read as "not A").

For the two occupancy cases of a cell, o (the cell is occupied) and $\bar{o}$ (the cell is empty), and new information M (derived from a sensor measurement), the above definition creates the equation:

$$\frac{p(o|M)}{p(\bar{o}|M)} = \frac{p(M|o)}{p(M|\bar{o})}\frac{p(o)}{p(\bar{o})} \tag{1}$$

Now this can be rewritten as:

$$\frac{p(M|o)}{p(M|\bar{o})} = \frac{p(\bar{o})}{p(o)}\frac{p(o|M)}{p(\bar{o}|M)} \tag{1a}$$

Now suppose that there exists some information, M_1, that has already been processed into a map, i.e. $p(o|M_1)$ already exists and it is desired to integrate some new measurement, M_2, to find $p(o|M_1 \cap M_2)$. In order to make the analysis tractable it is assumed that the new measurement is independent from all previous information. This

may not be completely true, but for the purposes of constructing a map from many sensor inputs it simplifies the problem immensely. However, it is not implied that $p(M_1 \cap M_2) = p(M_1)p(M_2)$, since if M_1 indicates that the cell is occupied then it is hoped that M_2 would be more likely to indicate the same thing. Instead, what is meant is that, given that the cell is occupied, the probability of getting reading M_1 is independent of getting M_2, and similarly for the cell being occupied:

$$p(M_1 \cap M_2 | o) = p(M_1 | o)p(M_2 | o) \tag{2}$$

$$p(M_1 \cap M_2 | \bar{o}) = p(M_1 | \bar{o})p(M_2 | \bar{o}) \tag{3}$$

Another way to look at this assumption is that it is only assumed that the sensor's errors are independent from one reading to the next. This is especially true of noisy sonar sensor data, in which the errors vary greatly from one reading to another from the same sensor due to changes in range, orientation, etc. Combining this assumption with a single application of Equation 1a, results in:

$$\frac{p(o | M_1 \cap M_2)}{p(\bar{o} | M_1 \cap M_2)} = \frac{p(o | M_1)p(M_2 | o)}{p(\bar{o} | M_1)p(M_2 | \bar{o})} = \frac{p(o | M_1)p(o | M_2)p(\bar{o})}{p(\bar{o} | M_1)p(\bar{o} | M_2)p(o)} \tag{4}$$

We generally assume that the a priori probability of a cell being occupied is 0.5, i.e., $p(o)=p(\bar{o})=0.5$, so that the last factor in Equation 4 cancels out. When the information M_2 is a sensor reading, the value $p(M_2 | o)/ p(M_2 | \bar{o})$, for all cells and all possible readings, is called the sensor model. In other words, the sensor model is a function which attaches a number, $(p(M_2 | o)/ p(M_2 | \bar{o}))$, to every combination of sensor reading and cell location, relative to the sensor. This assumes that the sensor is isotropic in its world position and pointing direction. In general, the sensor model is a

function of the sensor reading, the location and orientation of the sensor, and which cell is being updated. Also, while the sensor reading M_2 may represent a continuous number indicating distance from the sensor, in general, each time the sensor is polled it will return an element from some set and M_2 will range over all elements of that set.

The sensor model is usually independent of the current map and can be stored in tables. A further speed up of the process can be achieved if the logarithm of the above probability ratio is used. In this case the model uses:

$$log \frac{p(o \mid M_1 \cap M_2)}{p(\bar{o} \mid M_1 \cap M_2)} = log \frac{p(o \mid M_1)}{p(\bar{o} \mid M_1)} + log \frac{p(o \mid M_2)}{p(\bar{o} \mid M_2)} \tag{5}$$

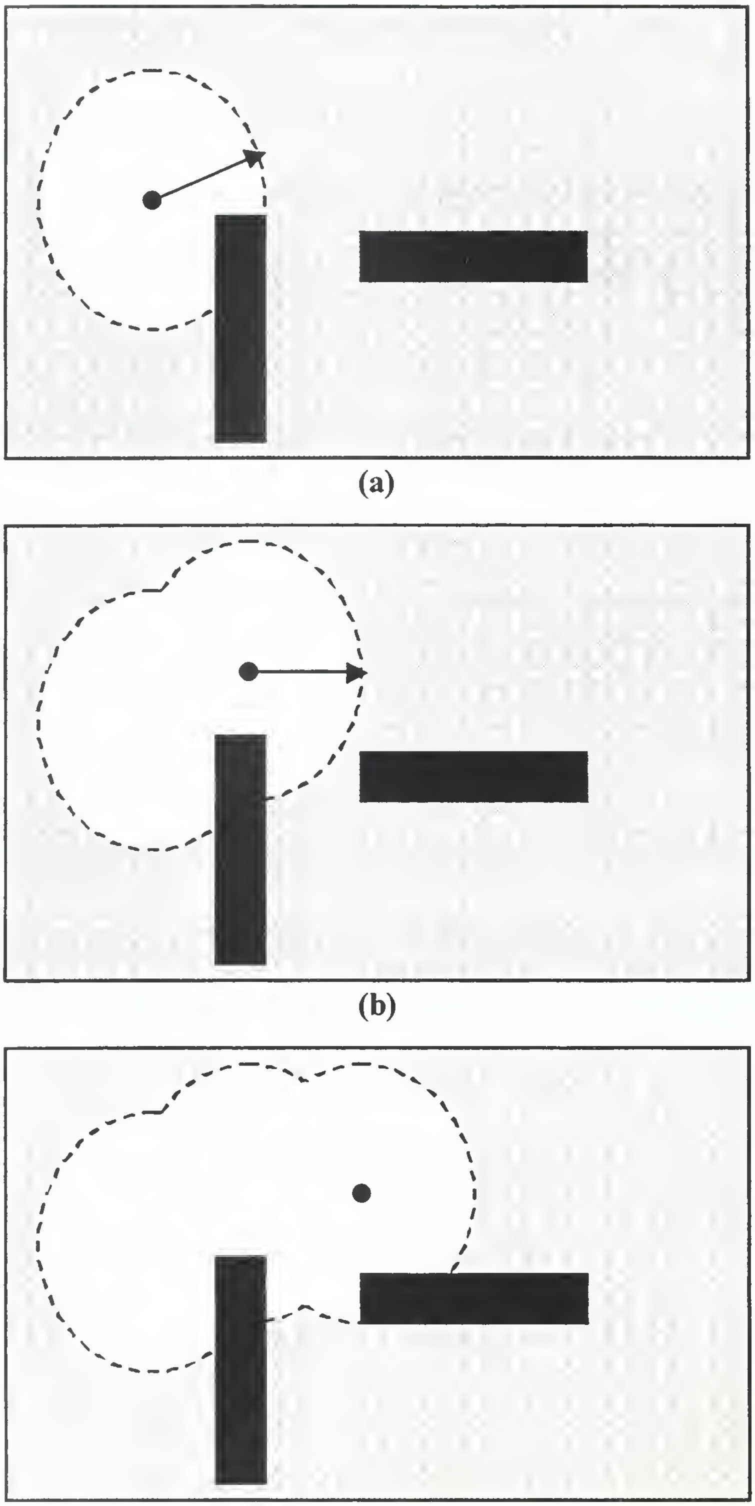
In the logarithmic method the combining formula is changed from a multiplication to a simple addition. In this case the logarithmic result itself can be considered as weight of evidence of cell occupancy. Therefore, the need for only a single addition per cell allows for very rapid updating of the evidence grid map based on the newly acquired sensor data.

IV. FRONTIER-BASED EXPLORATION

One of the goals of a robotic-based reconnaissance system is to reduce the amount of manpower required to reconnoiter an area. Any system worth building should be able to explore at least a limited area autonomously and in a fairly efficient manner. This means that the robots will have to be able to make use of the maps they will build as they move through their environment. Chapter III already discussed in great detail the way those maps might be represented, now it is time to discuss how a robot would use them to explore and map an area on its own.

A. DEFINITION

A robot exploring a new area will initially know nothing about that area except what it can detect in its immediate area with its own sensors. The limits of its sensors will form a boundary between known, explored space and unknown, unexplored space. Such a boundary is called a frontier. Within this boundary it is assumed that all obstacles are known and mapped. Thus, further exploration within this boundary would be futile. In order to maximize exploration in the least amount of time, the robot should move to the boundary between explored and unexplored space as soon as possible and use its sensors to expand the explored region. At the same time the act of expanding this known region will in turn create new boundaries or frontiers for the robot to explore. This is the central idea behind frontier-based exploration. This process is illustrated in Figure 8. [Ref. 22]



(a)

(b)

(c)

Figure 8. Example of frontier-based exploration. Robot begins by scanning immediate area, incorporates result into its map noting frontiers yet to be explored, moves to a frontier, and repeats the process.

Figure 8(a) shows a robot at startup in some unknown environment. The black circle represents the robot. The two large black rectangles represent obstacles in the area to be explored. The white space surrounding the robot is the area it can directly detect with its own sensors. The gray background represents the unexplored regions and the black dashed line represents the boundary between the known and unknown space.

In Figure 8(a) the robot can detect the top edge of the first obstacle, but the obstacle blocks its view what may be behind the obstacle. Accordingly, the robot chooses a frontier and proceeds to that frontier in order to continue the process of exploration and map building. Figure 8(b) shows the robot in its new position now able to see behind the first obstacle. After scanning in this position the robot moves on to the position shown in Figure 8(c) where it can now detect the second obstacle. This process could continue indefinitely as long as there are unexplored frontiers for the robot to explore.

B. FRONTIER DETECTION

Before a robot can decide on a frontier towards which to proceed in order to continue exploration, it must first decide where the frontiers are within the region it has already explored and mapped. In order to detect these frontiers a process similar to edge detection and region extraction in computer vision (also known as machine vision) is used to find the boundary between mapped open space and unmapped unknown space [Ref. 23].

Over the years many techniques have been developed in the area of computer vision to extract information from photographic images based on the pattern of changes in brightness in the picture [Ref 24]. Most of these techniques involve decomposing the image into a set of pixels, much like a grid. Each cell in the grid is given a value based on the brightness of the pixel that the cell represents from the original image. This grid is then searched for patterns that may indicate edges or patterns of interest.

There are a variety of methods for picking out the patterns in the image depending on the information that is desired. Some techniques involve the use of a set of "masks" composed of a small (on the order of 3 X 3 or 4 X 4) pattern of cells of varying values that are successively laid across the original image. As the mask is moved across the image the cells of the mask are convolved with the cells of the image beneath the mask and the resulting matrix indicated the presence of edges in that portion of the original image. Other methods rely on simple histograms of the brightness values of the cells in the original image and attempt to use curve-fitting techniques to pick out edges of objects in the real world.

Regardless of the methods used, these same types of techniques can be applied to detecting boundaries in frontier-based exploration. The same underlying grid format will be used as in computer vision, but in the case of frontier detection the values in the cells do not represent the brightness of a pixel. Rather, they represent the occupancy probability of a cell in the area the robot is exploring.

In general each cell in the area being examined for frontiers will be placed into one of three categories [Ref. 23]:

- **open:** when the current occupancy probability of the cell is less than the prior probability
- **unknown:** when the current occupancy probability of the cell is the same as the prior probability
- **occupied:** when the current occupancy probability of the cell is greater than the prior probability

A short explanation about the term "prior probability" is needed. When the evidence grid is first created it is necessary to initialize the cells in the grid to some value. Since at creation nothing in the grid has been explored it is logical to initialize all the cells to an occupancy probability value representing unknown, unexplored areas. All the implementations of frontier-based exploration discussed here [Ref. 22, 23, 25] initially set the cells' values to 0.5. When the frontier-detection process is first done it is this initial (prior) occupancy probability to which the current occupancy probability will be compared.

After a robot starts up or moves to new frontier it will make a sensor scan of the surrounding area. Based on the information returned from its sensors it will updated the occupancy probability value of any cell with direct range of its sensors. The robot will then use this updated information to perform frontier detection. Any open cell adjacent to an unknown cell will be labeled as a frontier edge cell. Adjacent edge cells are then grouped into frontier regions. Any frontier region above a certain minimum size (say

roughly the size of the robot) will be considered an accessible frontier and marked as such. [Ref. 23]

C. NAVIGATION

Once the robot has scanned an area and updated all cells of the evidence grid within range, it must now safely and efficiently navigate to a new frontier in order to continue exploration and map building.

1. Route Planning

Navigation to a new frontier should involve a path that is completely within known space, therefore, all obstacles in that space should be known. Route planning involves choosing the best path (based on some criterion such as length of path, nearest approach to obstacle, etc.) through the known space to the frontier to be explored. This path will be based on the latest update of information concerning explored space.

There are various algorithms such as depth-first, breadth-first, and A* search routines [Ref. 26] that attempt to search through a known map for safe and efficient navigation paths. However, a problem can arise if an obstacle (for example, another robot) moves into the chosen path since the last time information about the known space was updated. This may lead to the path the robot chooses to move to a new frontier being blocked. In that case, in order to avoid collision with the new obstacle and continue

navigating to the chosen frontier requires that the robot have some means of reactive avoidance.

2. Reactive Obstacle Avoidance

Reactive obstacle avoidance entails some method of detecting and reacting to mobile obstacles that appear in the robot's path that were not in the then-current map used by the route planning routine to plot a safe path for the robot. Mobile obstacles may include humans, other robots, or any of a number of other unpredictable phenomena that may exist in the robot's world. For the most part, reactive obstacle avoidance involves the use of relatively short-range sensors (IR, contact, etc.) or long-range sensors (sonar, vision, etc.) scanning in the area immediately surround the robot as it moves.

One important note about sensors and reactive obstacle avoidance is that the required scanning rate of the obstacle avoidance sensors is closely related to the expected travel rate of the robot and the anticipated characteristics of the area in which it is traveling. Obviously a quickly moving robot in an area where new obstacles appear frequently will need to scan the local area around it much more frequently than a slow moving robot in a well known, stable area.

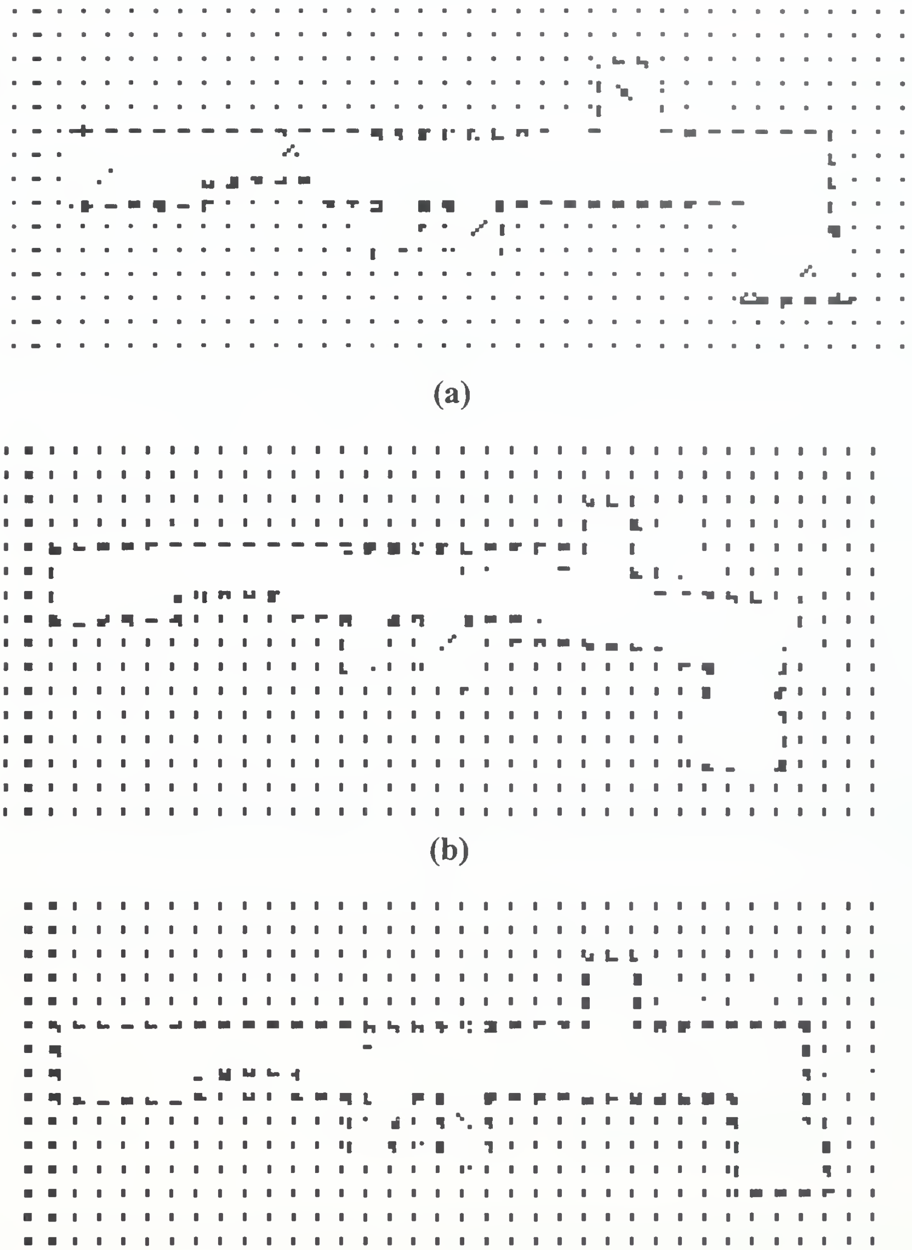
There are several different actions that a robot might take upon detecting a new obstacle along its planned path. One common method is to use some sort of very low-level navigation routine that simply finds a path around the new obstacle and gets the robot back on the previous planned path as quickly as possible. This eliminates the need to call on the slower full route-planning algorithm. For small obstacles in the robot's

planned path this is usually the method used. However, a problem can arise with this method when the new obstacle is so large that the low-level process cannot easily find a way for the robot to get around it. Normally, in this case the robot would stop and the full route-planning algorithm would be called on to find a new path to the robot's destination based on the new information about obstacles in the robot's path.

3. Localization Error

Every time the robot moves there is some slippage of the wheels that will cause the odometric encoders to incorrectly record the distance and direction the robot has traveled. Eventually, without some means of correction, the localization errors become so great that mapping and navigation become impossible. Methods of minimizing localization errors in mobile robots are the topic of much research [Ref. 20, 22, 27].

Figure 9 demonstrates the effects of robotic mapping with and without localization error correction methods while mapping a long, obstacle filled hallway. Figure 9(a) shows an evidence grid representation of the area to be mapped. Figure 9(b) shows a map developed by a frontier-based exploration system without the aid of any localization error minimization techniques. As the robot's coordinates become uncertain the sensor return data begins to "drift." Figure 9(c) shows the map which was developed using a continuous localization process that seeks to correct for dead reckoning errors that accumulate as the robot moves throughout the area to be explored. [Ref. 22]



(a)

(b)

(c)

Figure 9. Example of localization error and correction. Shown from top to bottom are the ground truth evidence grid, the map constructed without localization, and the map constructed with localization. [From Ref. 22]

D. NRL IMPLEMENTATION ON SINGLE NOMAD 200 ROBOT

The full implementation of the frontier-based exploration code (for both single and two robot systems) as developed at NRL consists of over 60 separate C and C++ routines that are then compiled into a single process. Many of these routines handle the various 'housekeeping' functions of any large, complicated program (i.e. display, input/output, etc.) and do not bear directly on this thesis. Relevant routines and their use will be discussed below and portions of the code from these routines will be reproduced in the appendices.

The latest version of the full code (as well as all previous versions) is available from Brian Yamauchi (yamauchi@aic.nrl.navy.mil) at NRL's Navy Center for Applied Research in Artificial Intelligence (NCARAI). The NPS modified version of the code is available from Xiaoping Yun (yun@ece.nps.navy.mil) at the NPS Department of Electrical and Computer Engineering (ECE).

1. System Overview

There are a few major processes in the NRL code that bear directly on single robot, frontier-based exploration. The three key parts of their single-robot system that are relevant here are: the use of laser-limited sonar (LLS) for sensor scanning at new frontiers, the exploration routine used for the detection of new frontiers and the subsequent movement to and scanning of those frontiers, and the integration of the new scan with the robot's current map.

2. Laser-Limited Sonar (LLS)

There has been much study of the characteristics of the simple type of Polaroid sonar units found on the NOMAD 200, the NOMAD SCOUT, and many other research and commercial robots [Ref. 28]. The low cost, low weight, and low power consumption of these types of sonars has made them very popular among robot builders and designers, however, they do have their drawbacks. As noted in Chapter III sonar returns in the real world environment tend to include much noise and many extraneous or false returns. Many of these questionable sonar returns are caused by a phenomenon known as specular reflection.

Specular reflection occurs when a sonar pulse hits a flat surface at an oblique angle and reflects away from the sensor instead of directly back to the sonar detector [Ref. 25]. When this happens there may be several different results depending upon the circumstances. If the sonar pulse reflected off of the oblique surface encounters a flat reflective surface soon after, then the detector may still get a return, but the first object that the sonar pulse struck will appear to be further away than it is in actuality. If the sonar pulse continues on to the sonar's maximum range without striking another object, then the nearby object may not be detected at all, but instead it will appear that there is a large open space surrounded by an unknown area.

In reality, the possibility of not detecting a nearby obstacle is not as great as it first appears. With many sonars onboard the robot firing from several different angles, there is usually not much problem with getting some measurable level of sonar from

nearby objects and detecting them even if there are facing obliquely to some sensors. However, the problem of detecting "phantom" obstacles and false open spaces is still a problem that needs to be considered. Figure 10 provides an example of a normal sonar return and two of the possible results of a specularly reflected return.

Figure 10(a) illustrates a normal sonar return with the sonar pulse represented by the ray emanating from the sonar, striking the nearby obstacle, and a measurable amount of energy returning to the detector. The head-on encounter of the sonar pulse with the flat face of the obstacle toward the sonar provides for a clear return path. In Figure 10(b), however, the oblique angle of the nearby obstacle reflects the sonar pulse away where it encounters the second obstacle. If a measurable amount of energy is returned from the second obstacle, then it may appear to the sensor that there is a "phantom" obstacle at the point shown in the figure. There may be a partial sonar return from the first obstacle as well, which could further confuse the sensor about the exact nature of nearby obstacles.

Figure 10(c) demonstrates what may happen if there is no second obstacle within the maximum range of the sonar sensor after the pulse has been reflected from the first nearby obstacle. The sensor would receive either a very weak or non-existent return from the nearby object. If no return is received then it will appear that a large open space exists in the area shown, when in fact there is really no information known about that area at all. Since this false open region would most likely be surrounded by unknown space it would also have the effect of creating false frontiers for the robot to explore.

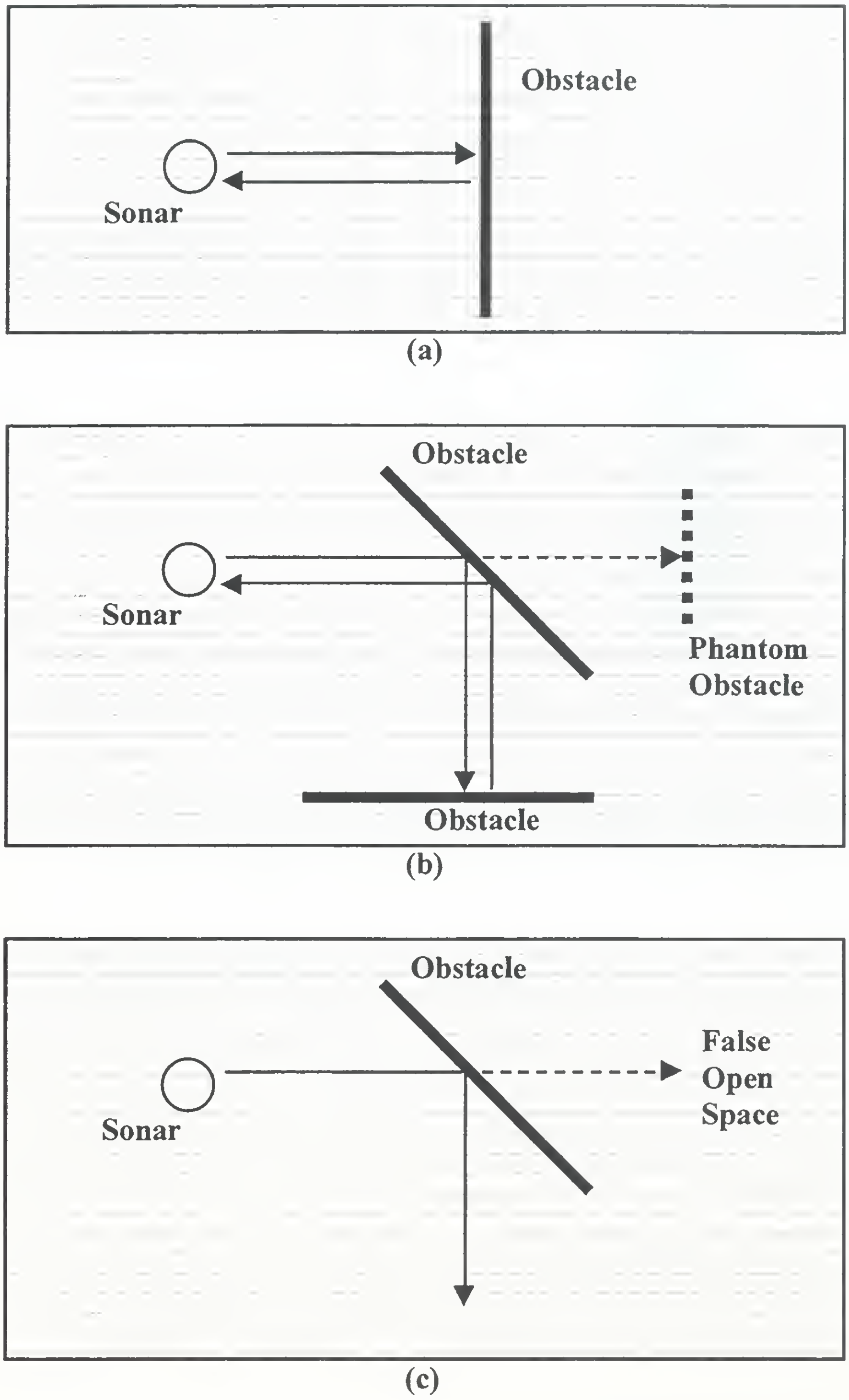


Figure 10. Examples of a normal sonar return, a specularly reflected sonar return that creates a phantom obstacle, and a specularly reflected pulse that generates a false open space. (After Ref. [25])

In addition to the sonar sensors, the NOMAD 200 has a laser based range finding systems that does not suffer from these same type of specular errors. The researchers at NRL have taken advantage of this and created a technique known as laser-limited sonar (LLS). By using the readings from the laser rangefinder in combination with the readings from the sonar it is possible to eliminate most many false readings from walls and other large obstacles that cause the majority of specular reflections. If the laser returns a range to obstacle less than the sonar, then the evidence grid is updated as if the sonar had given the range indicated by the laser. [Ref. 23]

The laser cannot be used exclusively for mapping because the laser rangefinder currently available on the NOMAD 200 only operates in a two-dimensional plane, while the sonar senses obstacles within a three-dimensional cone radiating out from the robot. Objects above or below the plane of the laser will be missed by the laser, but detected by the sonar. Figure 11 provides an example of how this might happen. In this figure the laser plane is above the obstacle and thus the laser rangefinding system never detects it, but the sonar cone emanating from sonar sensor does intersect the obstacle. This is a case of using two different sensor types that compliment one another. A three-dimensional rangefinder would be an alternative that would combine the best aspects of both sensors, but presently these type of systems are too large, expensive, and power consuming to be commonly used on mobile robots. [Ref. 23]

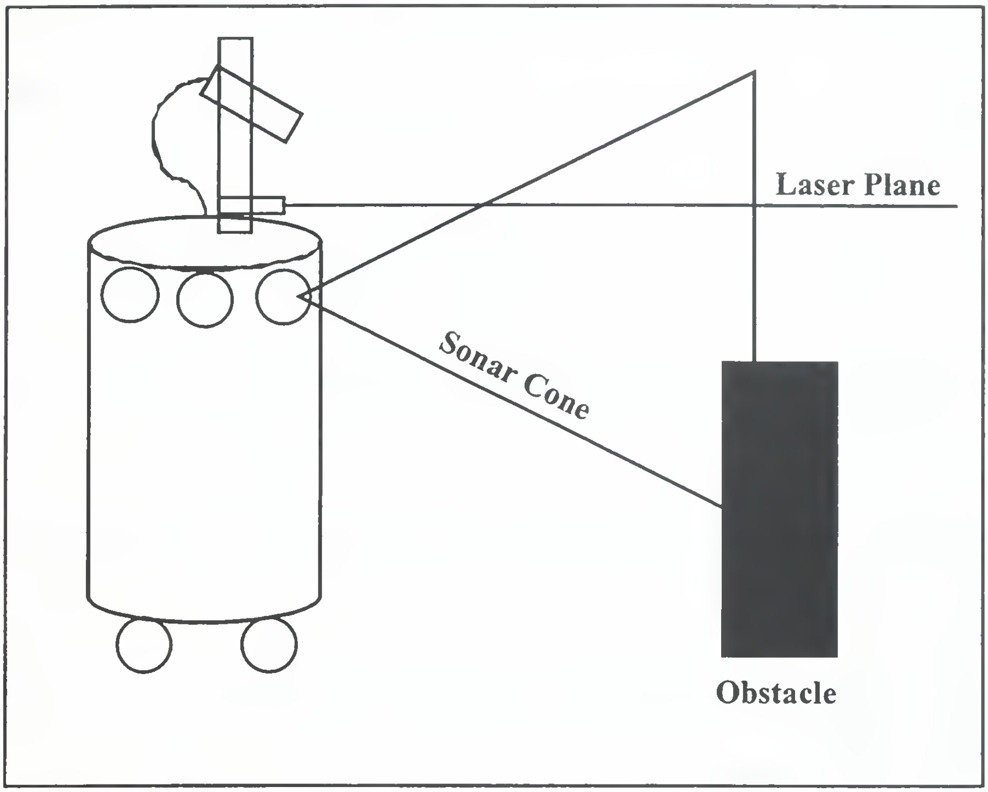


Figure 11. Example of two-dimensional laser rangefinder failing to detect an obstacle that is within the detection cone of the sonar sensor. (After Ref. [25])

3. Frontier-Based Exploration Routine

The heart of the frontier-based exploration code is in the file *agent.cc*. It is in this operation that the frontier-detection, navigation, and exploration behaviors are described. This procedure also controls the laser-limited sonar scanning technique mentioned above and also takes care of integrating the newest scan with the current map via the method described below. The process begins by completing an initial sensor scan of the area upon startup and using the data obtained to construct an initial evidence grid map. After the initial map is created, a frontier detection subroutine is called to find and note nearby frontiers for further exploration. Once the frontier detection is complete the exploration

and navigation subroutines are called to choose the robot's next destination and get it there safely.

a. Frontier Detection Subroutine

The method of frontier detection for this initial map and all subsequently generated maps involves using the edge detecting techniques described above on the most current evidence grid map that the robot has at that time. Mapped obstacles or edges separate frontier regions. The centroid (roughly the center of a non-symmetric region) of each frontier region is marked as the robot's target destination for exploring that region.

Figure 12 illustrates the different parts of the frontier detection process. Figure 12(a) demonstrates an evidence grid built by a robot in a hallway next to two open doors. Figure 12(b) shows the frontier edge segments detected in the evidence grid by the frontier detection process. Figure 12(c) shows the frontier regions that are greater than some threshold value (in this case roughly the size of the robot). The centroid of each of the frontier regions is marked with a crosshair and numeric label. The frontier regions labeled 0 and 1 represent the open doorways and the frontier region labeled 2 is the unexplored portion of the hallway. [Ref. 22]

b. Exploration Subroutine

Once the frontier regions have been found on the most current evidence grid map, the robot must decide which frontier to explore next. The path planner in the exploration code uses a relatively simple depth-first search on the evidence grid. It starts

at the robot's current position and attempts to select the shortest obstacle-free path to the centroid of the frontier region chosen as it next destination. [Ref 23]

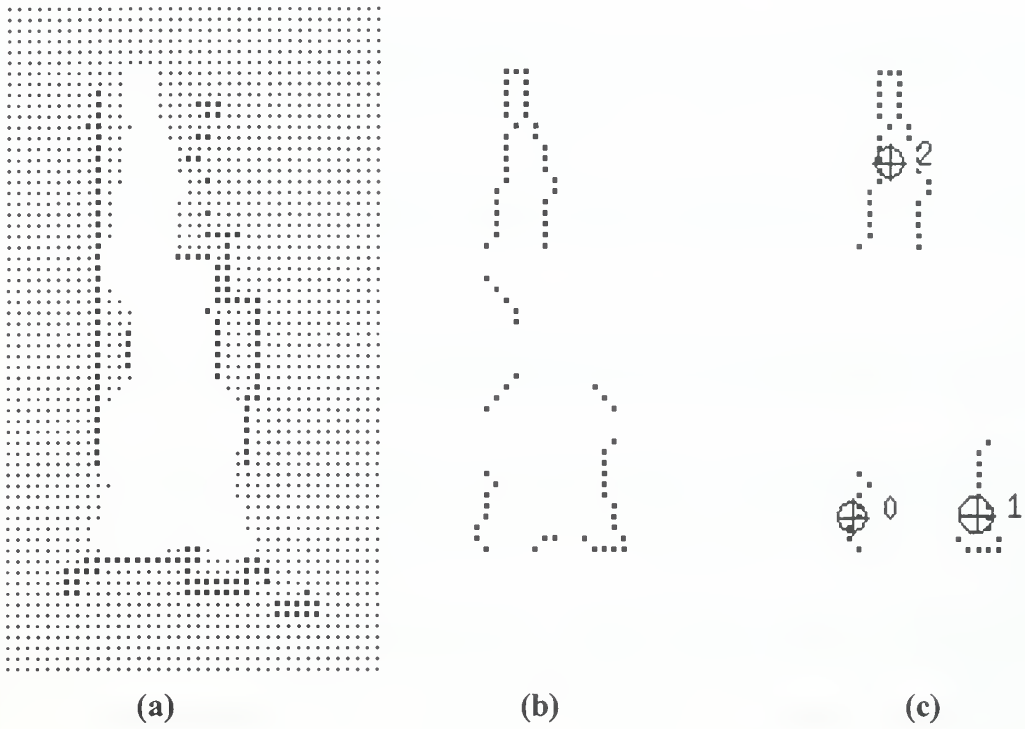


Figure 12. Example of frontier detection. From left to right: the evidence grid constructed, the frontier edge segments, and the frontier regions with the centroid of each region labeled. (From Ref. [22])

c. Navigation Subroutine

Once the exploration subroutine has chosen a frontier to explore it is the goal of the navigation subroutine to get the robot to its intended destination in a safe and efficient manner. Using the path chosen by the depth-first search and a variety of reactive obstacle avoidance behaviors the navigation subroutine guides the robot. The navigation method used in the exploration code is sufficient to allow the robot to steer

around small obstacles in order to get back on the pre-selected path. If the robot becomes blocked or for some other reason cannot make any progress toward its destination, then after a certain amount of time, that location will be added to the list of inaccessible frontiers that the robot cannot reach. At that point the robot will conduct another sensor scan, update its current evidence grid map, and attempt to navigate to the next accessible, unknown frontier as chosen by the exploration subroutine.

4. Integrating New Scan with Current Map

The map integration routine uses the method described in Chapter III for fusing the new sensor data onto the current map. The frontier-based exploration process uses a modified version of the log-odds Moravec code described in [Ref. 17]. In the NRL code, each cell of the evidence grid is assigned a value from –127 (definitely empty) to +127 (definitely full) for its occupancy value.

When the sensor fusion procedure described earlier is used to combine the current map data and the new scan data, the result will be a new map that reflects the effects of the most current sensor data. Similar data between the current map and the new scan will tend to reinforce one another driving well-mapped cells toward a floor value of –127 or a ceiling value of +127. Likewise, if the data in the new scan conflicts with the data in the current map, occupancy values of those cells will be driven toward smaller absolute values with zero representing a cell whose occupancy is completely unknown.

There is one important thing to note about coordinate systems used in the exploration process. When first initializing the robot, the user is asked to enter the

robot's X, Y coordinates and an initial orientation angle. For single robot exploration this is not necessarily required, as the robot could assume that its starting position and orientation are 0, 0 and 0 degrees and build a map around that point. However, for multiple robot exploration the robot knowing its starting position becomes very important as all maps sent to other robots are referenced to the shared global coordinate system. This will be discussed in more detail in the following chapter.

E. NPS IMPLEMENTATION ON SINGLE NOMAD SCOUT ROBOT

In order to use the frontier-based exploration code developed for the NOMAD 200 on the NOMAD SCOUT many modifications to the original NRL code are necessary. Throughout the original code there are many obvious, as well as hidden, dependencies and assumptions based around the sensor suite and mobility base of the NOMAD 200. Most of the modifications can be broken down into one of two categories: those changes due to the differences in the movement commands between the NOMAD 200 and the NOMAD SCOUT and those changes due to differences in the available sensors on the two platforms.

1. Mobility Modifications

As described in Chapter II, the NOMAD 200 has a three-wheel synchronous drive system with a turret that can rotate independently of the base, while the NOMAD SCOUT is a two-wheeled, two-degree of freedom differential drive robot which has no

turret. The NOMAD SCOUT was the first platform from Nomadic Technologies (manufacturer of all the NOMAD hardware and software products) that was built around such a mobility base. All their software prior to this had been designed for a three-wheel synchronous drive system with an independent turret. Early adopters of the NOMAD SCOUT, such as NPS, are using modified versions of the standard NDHE to interface with the new platform. Nomadic is currently developing a much more advanced version of the NDHE that will be more flexible in terms of working on multiple types of robots with varying mobility and sensor capabilities.

Fortunately, Nomadic has developed a set of macros that accept differential drive commands for the NOMAD SCOUT and transforms them into equivalent synchronous drive commands that the software understands. These modified movement commands convert the decoupled translation and steering commands used by the NOMAD 200 into differential drive values required for the NOMAD SCOUT. So when the software is controlling NOMAD SCOUT it is actually modified synchronous drive commands that are being sent to the robot. To eliminate any software conflicts due to the lack of a turret on the NOMAD SCOUT, the conversion macros send a null (zero) value in place of any turret rotation commands to the robot. [Ref. 10]

2. Exploration and Navigation Modifications

The exploration and navigation modifications from the NOMAD 200 to the NOMAD SCOUT are much more extensive than the mobility modifications. The assumptions about the sensor suite of the robot designed into the original frontier-based

exploration code make for a large number of modifications to work on the NPS research platform.

a. Elimination of Laser-Limited Sonar Dependency

In the original frontier-based exploration process the laser-limited sonar technique described above is used to scan whenever the NOMAD 200 reaches a new frontier. Since the laser rangefinding system is fixed in place on the NOMAD 200 turret, this involves rotating the turret a complete 360° while the base of the robot remains in place. There is also an option to use only the sonar sensors to scan at new frontiers. If the sonars are being used as the only sensor, there is another option to rotate the turret through the 22.5° arc that separates the sonar sensors and take sonar readings at intervals along that rotation.

In theory this gives the sonar sensors more opportunities to see obstacles from varying angles that might be less affected by specular reflection due to variations in the obstacle surface and what portion of the obstacle is struck by the sonar pulse. By modifying this sonar-only option to turn the entire robot instead of just the turret, it is possible to use this process on the NOMAD SCOUT in order to complete a sonar sweep upon reaching a new frontier. In addition, switching to the sonar-only option removes the dependency on a laser rangefinding system that the NOMAD SCOUT lacks.

b. Compensation for Inability to Timestamp Sonar and Pose Data

Originally it had been thought that once the laser-limited sonar dependency had been removed, that it might be possible to have the NOMAD SCOUT collect sonar

range data while on the move and eliminate the need to stop at frontier boundaries in order to collect new sensor readings. Taking sonar readings while moving can present a problem because sensor readings are not instantaneous and the robot's position or the environment can change significantly between acquisition and processing of the sensor data, thus causing the collected data to be inaccurate [Ref. 12].

At the speeds both the NOMAD 200 and NOMAD SCOUT typically travel this does not cause any difficulty for the reactive obstacle avoidance routine, but it can cause problems for the accuracy of the mapping routine. On the NOMAD 200 it is possible to attach the robot's pose information and a timestamp to every sensor reading so that data taken while the robot is moving can be correctly interpreted. The NOMAD SCOUT lacks this capability. In order to ensure that the sensor readings and pose information were as closely matched as possible it was necessary to break the sonar sweep at new frontiers (as described above) into small, individual movements. After each movement the robot was halted, the sonar readings were taken, and the sweep was continued. This has the effect of slowing down the overall mapping effort and the repeated small movements tended to increase the localization error.

c. *Specular Reflection Minimization and Side Effects*

In another one of the exploration code files (*grid.h*) it is possible to set the range at which sonar return data is considered "trustworthy." Because the NOMAD 200 has a laser rangefinder to confirm those sonar readings it is possible on the original code to leave this setting relatively high and disregard false readings. In the original code this

maximum sonar range is set at ten feet from the robot. Since the NOMAD 200 lacks a method of double-checking its sonar data this value is reduced to six feet. This also helps reduce errors due to specular reflection because it has been found that specular reflection errors are more prevalent at longer ranges [Ref. 28].

Unfortunately, there is an undesired side effect of reducing the "trustworthy" sonar range on the NOMAD SCOUT. With the decrease in range also comes a proportional decrease in the amount of new territory that is mapped whenever the robot reaches a new frontier. Mapping less area each time leads to an increase in the number of new frontiers to which the robot must travel in any given area to be explored compared to the number of frontiers with a longer sonar range. Increased travel leads to a faster buildup of localization errors when only dead reckoning from the robot's odometric encoders is used to determine the robot's position in the global coordinate system. Neither the original NOMAD 200 nor the current NOMAD SCOUT versions of the frontier-based exploration system incorporate any sort of localization error minimization process. Later work on the NOMAD 200 has included work with a continuous localization process [Ref. 20, 22] and it is hoped that this method or one like it may also be used on the NOMAD SCOUT in the future.

d. Reactive Obstacle Avoidance

While navigating to a new frontier, the reactive obstacle avoidance subroutine of the original exploration code uses the infrared sensors as well as the sonar sensors on the NOMAD 200 to detect nearby objects. The NOMAD SCOUT lacks

infrared sensors and it is necessary to remove any dependency on them and rely solely on the sonar sensors during the navigation and movement process.

It is also necessary to make some relatively minor modifications to routines that take into account the robot's size when determining if there is enough open space in a doorway or corridor for the robot to safely travel. The diameter of the NOMAD SCOUT is slightly less than that of the NOMAD 200 and thus it can move into a more constrained space.

V. MULTIPLE ROBOT INTEGRATION

Even a very capable and very well equipped single reconnaissance robot will still be restricted in the amount of area that it will be able to cover in a given time by the limits of its mobility base and the range of its sensors. In addition, a single robot reconnaissance system is very vulnerable in that a single failure on the one platform can have catastrophic consequences on the ability of the system to perform its mission. By combining multiple robots together into a single integrated reconnaissance system, it is possible to have a greater area of coverage in a given time, quicker coverage of a given area, and continuous or overlapping coverage of high value target areas of interest. In addition, the use of multiple robots provides for a graceful degradation, rather than failure, of the system if individual robots fail to perform for some reason. In Chapter IV the mechanics of a possible single-robot exploration system were discussed. In this chapter the dynamics of using multiple numbers of such robots will be explored.

A. CENTRALIZED VERSUS DISTRIBUTED CONTROL

There are two differing philosophies concerning command and control of multiple robot systems. The centralized approach advocates some sort of supervisor or controller process that receives inputs from the individual robots and provides information and commands back to them. The distributed approach calls for processing sensor data at the local level and individual robots making autonomous "decisions" based on that information. Between these two methods there is a wide range of combinations and

permutations depending on the intent of the system designers and how they choose to implement the system.

In the case of extremely centralized control there may be very little on-board "intelligence" on any of the individual robots and all commands of any sort (including motor and actuator commands) may have to come from the supervisor process. In this case the individual robots are little more than remote sensors on a mobility platform teleoperated by a central controller. The advantage of such a system is that each individual platform may be cheaper per unit. The main disadvantage is that highly centralized control leads to a single point of failure for the entire system and it will most likely also have a high bandwidth requirement if all raw sensor data and motor commands are required to be sent over the air [Ref. 29]. Another, less centralized, system may allow for centralized collection of processed sensor data from the individual robots which is then sent out to all the robots which then independently choose their own destinations for further exploration. Yet another system may call for the supervisor process to explicitly designate where individual robots will travel to and what tasks they will perform.

In a fully distributed system each individual robot might operate completely autonomously of the rest of the system. More autonomy on individual robots can lead to increased complexity and unit cost per platform, but it also allows elimination of the single point of failure problem that plagues highly centralized systems. Autonomous operation does not preclude cooperative effort between robots, but without a central supervisor it can complicate the problem of coordination. Lack of coordination in a

distributed system may lead to decreased efficiency and possibly even counterproductive behavior on the part of individual robots in respect to the goals of the system. This will be discussed in more detail in Chapter VI.

B. SENSOR FUSION

The evidence grid based mapping technique as described in Chapter III provides a good basis for the integration of sensor data from multiple, geographically separated robots. Sensor readings from different robots are fused in the same manner that sensor readings taken from a single robot at multiple locations are fused together. All that is required is that the sensor readings be referenced to a common global coordinate system in order for the sensor data from multiple locations to be properly correlated.

The details of how and where the sensor fusion is done will vary depending on the organization of the rest of the system. In a centralized system the coordinator/controller might receive all the individual robot sensor readings, fuse the data into a new global map, and redistribute that information throughout the system. A more robust distributed system might allow for each individual robot to receive all the other robots sensor readings (or at least those nearby), perform the sensor fusion locally, and "decide" for itself where to explore next based on some given criteria. Other implementations might allow for limited local processing of sensor data at the individual robot level with the resulting details transmitted to a remote higher level supervisory process.

C. COOPERATIVE EFFORT

In order to maximize the efficiency of a multiple robot system there has to be some degree of cooperative effort on the part of the individual robots. There are many possible degrees of cooperation as well as a multitude of methods and means of implementation. Both communication and coordination may be explicit or implicit with many varying combinations in-between.

1. Explicit Communication and Coordination

An explicit communication model allows for directed communications between each individual process and every other individual process as well as to and between any controlling or supervisory processes as well. While this model allows for a high degree of coordination and control, it can also become a communications nightmare very rapidly. The number of required links, L, for N separate processes in a fully interconnected system is given by:

$$L = \sum_{i=1}^{N-1} i \tag{1}$$

Another variation of this communication model is to use a few, or even just one, common broadcast channel(s) and then to attach some form of addressing to each message sent. Each individual robot or process then listens to the common channel(s) for messages addressed specifically to it, ignoring all others. This provides much the same functionality of the totally interconnected model with much less communications complexity.

Explicit coordination involves the passing of directions or orders, as compared to simply information, from a process outside the individual robot in order to influence or direct the robot's actions. It removes a degree of autonomy from the individual robots (which may no longer have a choice about their tasks and behaviors) and moves the decision-making ability to a higher level. Explicit coordination is usually associated with a hierarchical organization, but it can also be implemented on a peer to peer level where all the robots may be "equal," but at least on a temporary basis one robot may be able to direct the actions of another [Ref. 30].

Explicit communication and coordination is exemplified by the process of a parking lot attendant directing cars into and out of the parking lot. The attendant is explicitly communicating with each of the vehicle drivers through a series of hand and arm gestures (often accompanied by verbal expressions). The attendant is also providing explicit coordination amongst all the vehicles and has a direct line-of-sight communication channel with each driver.

2. Implicit Communication and Coordination

Implicit communication calls for processes not to pass information directly to another process, but to still convey information in some manner to interested parties. This may involve some sort of display or broadcast on the sender's part, or the process or robot may have some sort of noticeable behavior that an outside observer can interpret [Ref. 31].

In general implicit communication has lower direct communication requirements from robot-to-robot or robot-to-supervisory process. However, there is a corresponding increase in the requirement for an individual robot to be able to interpret other robots' behaviors or displays and extract useable information them. This requirement may have a great effect on the unit cost per robot depending on the complexity of the information implicitly passed from robot-to-robot or robot-to-supervisory process.

Implicit coordination involves a single robot interpreting the actions and behaviors of other robots in the environment around it and taking individual action in accordance with the general goals of the system. This calls for a higher degree of autonomy on the part of the individual robot in order for it to know when to do the "right" thing at the "right" time. Implicit coordination is generally associated with a peer-to-peer organization where all robots are "equal," but it can also be implemented in a hierarchical structure with "lower" robots taking appropriate cues from the behavior of "supervisor" robots and vice-versa [Ref. 32].

Implicit communication and coordination is perhaps best exemplified by the process of an audience leaving a theater at the end of a movie or play. Generally, there is no overarching supervisor directing people into line and out of the theater and people do not explicitly announce their intentions to move into line to those around them. Instead, each person observers the actions of those around him/her and making a decision on when and where to move based on their actions and behaviors and following the general goal of leaving the theater.

D. NRL IMPLEMENTATION ON TWO NOMAD 200 ROBOTS

The interprocess communications routines used in the frontier-based exploration code were developed by Bill Adams (adams@aic.nrl.navy.mil) at NRL's NCARAI facility. The NPS modified versions of these routines are available from Xiaoping Yun (yun@ece.nps.navy.mil) of the NPS ECE Department as part of the NOMAD SCOUT modified frontier-based exploration code. Relevant routines and their use will be discussed below and portions of the code will be reproduced in the appendices.

1. System Overview

There are a couple of processes in the NRL code that bear directly on multiple robot, frontier-based exploration. The two key parts of their two-robot system that are relevant here are the communications process itself and the process of a robot integrating another robot's map with its own.

2. Communication Process

In the NRL code for two-robot, frontier-based exploration information about the world is shared, but each robot maintains its own map and makes its own decisions about where to navigate [Ref. 25]. There is no higher level supervisory process directing the individual robots or coordinating their actions. Normally this would be thought of as implicit communication and coordination. However, the system has to make use of an

explicit communication architecture to emulate the implicit communication process due to the limitations of the available networking protocols.

Even thought conceptually the process that is modeled is a peer-to-peer relationship the limitations of using existing TCP/IP networking tools preclude the system actually implementing this model. Instead, the original code simulates a peer-to-peer relationship using a standard client-server model. In the two-robot code the first robot is always designated as the server while the second robot is always the client.

Also, even though the model of the original code implies implicit communication, messages concerning new map availability are actually passed explicitly from robot to robot. It is also important to understand that in both the NOMAD 200 and the NOMAD SCOUT implementations that there is no controlling process actually running on the robots themselves other than low level motor controls in the robot's firmware. The controlling process of the robot is running on a remote UNIX workstation and all "communications" between robots is actually communication amongst the remote controlling processes. Also each controlling process is a client to the *Nserver* process. Figure 13 provides an illustration of how this is all connected together for a two-robot system.

As shown in Figure 13, all three processes, the *Nserver* and the two mobile robot processes, are running on the same UNIX workstation. In reality these three can all be on the same or separate workstations or any combination in-between as long as there is a shared memory location to which each robot process can write the map it will share with the other robot processes.

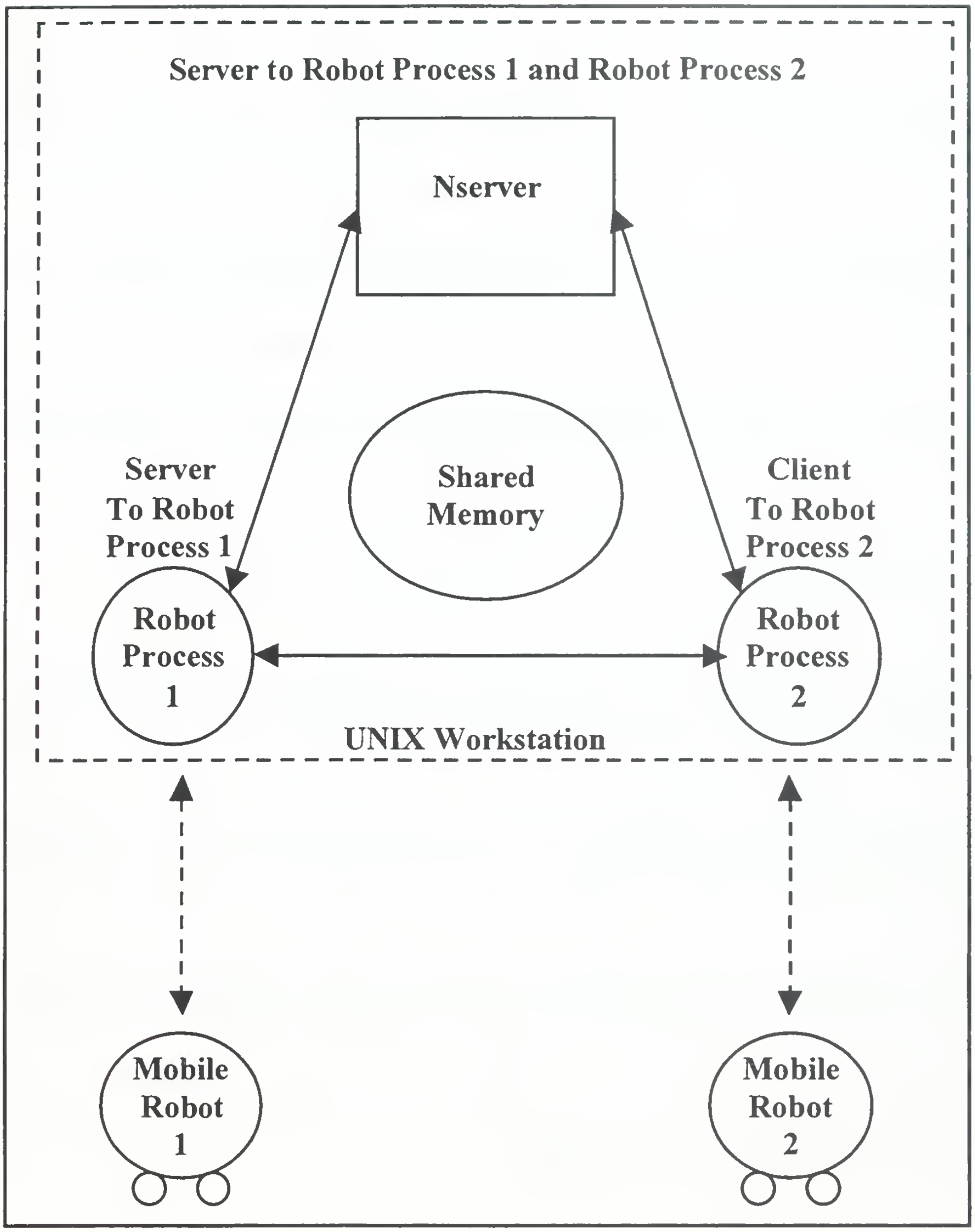


Figure 13. Illustration of the communication process used for the two-robot frontier-based exploration system.

During the exploration process whenever the individual robot completes a sensor sweep at a new frontier, its controlling process writes the result of that local scan into the shared memory location. This file (*local1.eg* or *local2.eg* depending on which robot process that creates it) is an evidence grid representation of the most current local map of

the area surrounding the robot of the controlling process. In the original representation this file is not the full global map maintained by each robot process.

After the robot process has finished writing the updated local map to the shared memory location it sends a message to the other robot process that there is a new local map available. This is a case where an explicit communication is used to simulate what would normally be a broadcast if allowed for by the network protocols being used. Instead of the updated map data itself being sent, there is a directed message to the other robot process transmitted.

After the update message has been sent to the other process the controlling process checks to see if it has received its own message from the other process indicating that a new local map file is available. If there is one available it reads it from the shared memory location and proceeds to integrate that new remote local map into its current global map. Using this method writing new maps, sending messages to other robot processes, and checking for remote local maps to integrate is only done after a new sensor sweep has been completed at a new frontier.

One of the drawbacks to this method is that it is possible for a robot process to miss a remote map update from another robot process. This can happen if a robot process is directing its associated robot on a long traversal from one frontier to a new one to be explored. It is possible for the other robot process to explore a frontier, write an updated map, move its robot to a new frontier, explore the new frontier, and overwrite the previous update all before the other robot reaches a new frontier and completes its exploration, writing, and remote map reading routines.

The original information is lost because the local map file that is written to the shared memory location only covers the immediate area around the robot. Once the robot moves to a new frontier and a new local map file is written it is very unlikely that there will be any map area overlap between the new map and the previous one. This can lead to a robot process making decisions on where next to explore based on incomplete information as to where other robots have already explored.

3. Integration of Foreign Maps

The integration of a foreign or remote local map uses the same methods described previously in Chapters III and IV for fusing new sensor data onto the current map. The sensor data from a remote map is fused with the robot process's global map in the same manner as if the process's associated robot had collected the same data itself. The important thing to note is that the remote local map must also have attached to it the global X and Y coordinates where the data was collected so that it may be registered correctly during sensor fusion with the global map.

E. NPS IMPLEMENTATION ON FOUR NOMAD SCOUT ROBOTS

In order to use the communications routines developed for the two-robot NOMAD 200 frontier-based exploration code on a greater number of NOMAD SCOUT robots a few modifications to the original NRL are necessary. None of the changes are actually specific to the NOMAD SCOUT so the modifications made here will work just

as well on an increased population of NOMAD 200 robots. It is hoped that other researchers with larger numbers of NOMAD 200 robots will be able to make use of the modified code developed at NPS. The modifications can be broken down into two parts: extending the client-server architecture to mange more than two robot processes and transmitting the global map vice the local map from the server robot process.

1. Extended Client – Server Model

There are a variety of possible different approaches to extending the original code's client-server model for two robots to a client-server model for greater than two robots. In order to maintain the fully interconnected architecture of the original NRL code it would be necessary for individual robot processes to function as both clients and servers depending on the circumstances. An example of how this might work for four robot processes is shown in Figure 14. A process being both server and client simultaneously is not without precedent as all the robot processes are also clients to the *Nserver* and the first robot process is also a server to the second robot process in the original model.

However, as seen in Figure 14, the number of interconnections increases rapidly as the robot process population grows. It becomes apparent that this is an unnecessarily complicated and unwieldy implementation for a large number of robot processes and an alternative method is used that while not fully interconnected between robot processes, still provides suitable connectivity for the transmission of remote map file information.

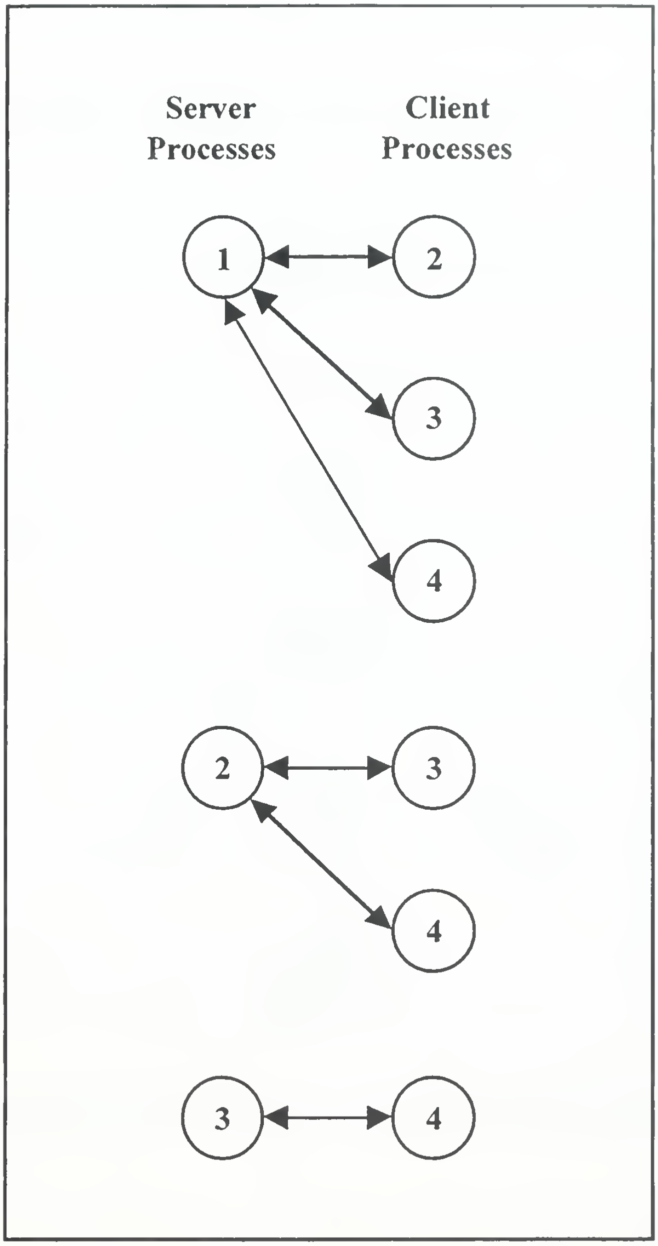


Figure 14. Illustration of a fully interconnected communications architecture for four robot processes using a client-server model.

Instead a single robot process is used as a server and all the other robot processes are clients to that single server process. This model is shown in Figure 15. As in the original NRL code the first robot process acts as the server. Also, it should be noted that all the robot processes remain clients of the *Nserver* program. This approach eliminates

the full interconnection of the original NRL model, but greatly simplifies the modification of the code to work for larger numbers of robots. However, without the direct connection from client process to client process there is a need for another way to transfer remote local map information between client robot processes. The solution to this problem is discussed in the next section.

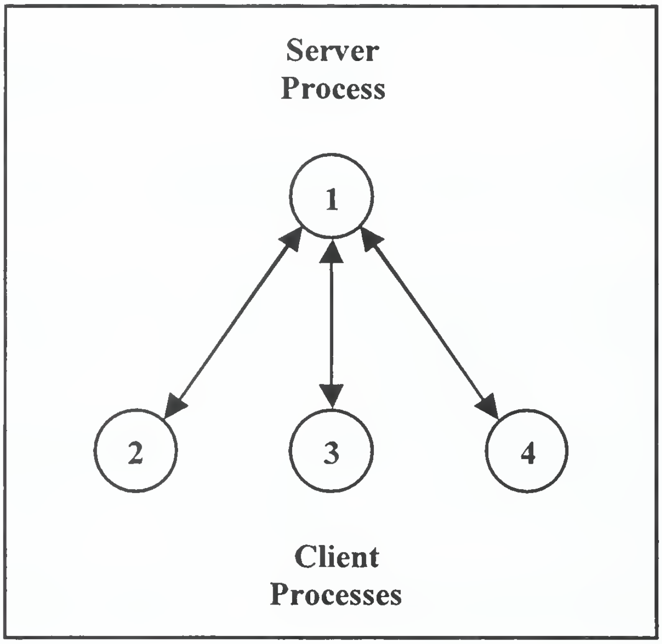


Figure 15. Illustration of the communications architecture implemented for four robot processes using a client-server model and retaining the single robot process as server.

In this model each of the client processes is in effect part of a two-robot system, itself and the server process. Only the server process "sees" all the individual client robot processes. However, there is still a good deal of robustness to the system. If one of the client process robots fails the rest of the system will continue to function in the same manner with the smaller robot population. If the server process fails there will be no more sharing of map data, but each robot process will continue to explore individually until there are no more frontiers remaining.

2. Transmission of Global vice Local Maps from Server

In the original NRL code whenever a robot process writes a map file to the shared memory location it is a local map file of just the immediate area surround the associated robot of that process and showing sensor data from the latest sensor sweep only. It is possible to modify the code so that instead of just writing the local map data, that the global map maintained by that process is written to the map file. This global map file incorporates all the separate local sensor sweeps that that process has made up to that time as well as any remote map files that it has fused into its global map.

Modifying the code in this manner makes it possible for each robot to write the entire evidence grid representation of its global map to the shared memory location (*globalx.eg* where *x* is the number of the robot process). In this manner the server process incorporates the individual global maps of all the individual client processes into its own global map. When the client processes read the *global1.eg* remote map file written by the server process they then indirectly incorporate all the results of the mapping done by the other client processes.

This also solves the problem of individual sensor sweeps being “lost” due to a short move, explore, write cycle causing the file to be overwritten. In the case of writing the entire global map the new global map will incorporate the previous sensor sweeps as well. However, having all the separate robots write their individual global map files has an unexpected drawback as well.

When only the local sensor sweep information is propagated from robot process to robot process then "bad" mapping data from one robot (due to sensor errors, location errors, etc.) may be overwritten when another robot happens to scan the same area. Also in this manner temporary obstacles (people, other robots, etc.) are steadily eliminated or their positions updated on the global map that each robot process maintains. By having each robot send its entire global map an unwanted feedback loop for the reinforcement of "bad" data can occur.

If a client robot scans an area that happened to have a temporary obstacle or is very "noisy" due to specular reflection off of objects in that area, the results of that local scan will be incorporated into its own individual global map. Now when that global map is read by the server process it will fuse that data with its global map and write its updated global map to the shared memory location for all the client processes to read. After they read it and update their global maps, the next time they write their global maps for the server process they will also show the same "noisy" or temporary obstacle sensor data which will reinforce the previous data on the server process global map. The server process will in turn write this updated global map for the client processes to read and the feedback of incorrect or out-of-date sensor data will continue. This is not a desired situation.

In the final version of the NPS modified code only the server process writes its global map for the client processes to read. The client processes write out only the results of their local sensor sweeps for the server process to read. In this implementation temporary obstacle data or "noisy" sensor data will be propagated through the system

once when the server robot incorporates it into its global map, but it will not be reinforced by having that same sensor data sent back to it from the client processes. The tradeoff to this approach is that once again the server process may miss a client process local map if the server process is busy controlling its robot during a long movement between frontiers.

VI. RESULTS

Despite equipment delivery delays and the need for extensive software development, substantial initial testing of the NOMAD SCOUT multiple robot frontier-based exploration system at NPS was possible in the limited time available for research. Besides the results presented here, the major product of this research was a demonstrable robotic exploration system that will serve as a testbed for future projects involving both single and multiple robots. Presented here are the preliminary findings to date.

A. SINGLE ROBOT MAPPING EFFORT

Single robot mapping of a given area provided a baseline against which multiple robot mapping efforts were compared as well as ensured that the basic frontier-based exploration code and evidence grid map making routine functioned properly in conjunction with the NOMAD SCOUT robot. Early tests were also used to determine the best combination of map grid resolution, "trustworthy" sonar range, and given area to be explored that would yield optimal results for a single robot. The best combinations of these variables were then used for each individual robot in multiple robot mapping experiments.

1. Single Robot Test Conditions

The test area for all single and multiple robot trials was an approximately 37 by 37 foot research room with two large test benches that defined three major corridors in the

space. An additional test bench along one wall further constricted one of the corridors. Figure 16 is a simple illustration of the area with annotations for the various starting positions used in the trials. The center of the room was defined to be the origin of the coordinate system used in the map produced during the robot mapping trials. This origin is marked as position zero in the illustration below. The various corridors are referred to as top, middle, and bottom as labeled in the sketch of the room. Note the windows stretching along one wall, the metal cabinets, and the large number of doors. These were geographical features that proved especially challenging during efforts to map the area due to specular reflection effects.

Shown in Figures 17, 18, and 19 are a series of pictures taken of the test area in order to provide the reader with a better perspective of the environment. Note the large open spaces under the test benches. Because the lower portions of these benches were so close to the ground the sonar sensors often failed to detect them. It was necessary to fill in some of the space under the benches in order to enhance their sonar image before any worthwhile results were possible.

Figure 17 shows the top corridor of the test environment. The metal desks and windows in this area proved particularly difficult to accurately map. Figure 18 shows the middle corridor of the test environment. This corridor runs down the center of the test environment and the midpoint of this corridor served as the origin of the coordinate system used in all the mapping trials. Again, the windows at the end of this corridor caused difficulties in the mapping trials. Figure 19 shows the bottom corridor of the test

environment. The smooth-surfaced doors and the metal cabinet in this corridor were the major sources of mapping errors.

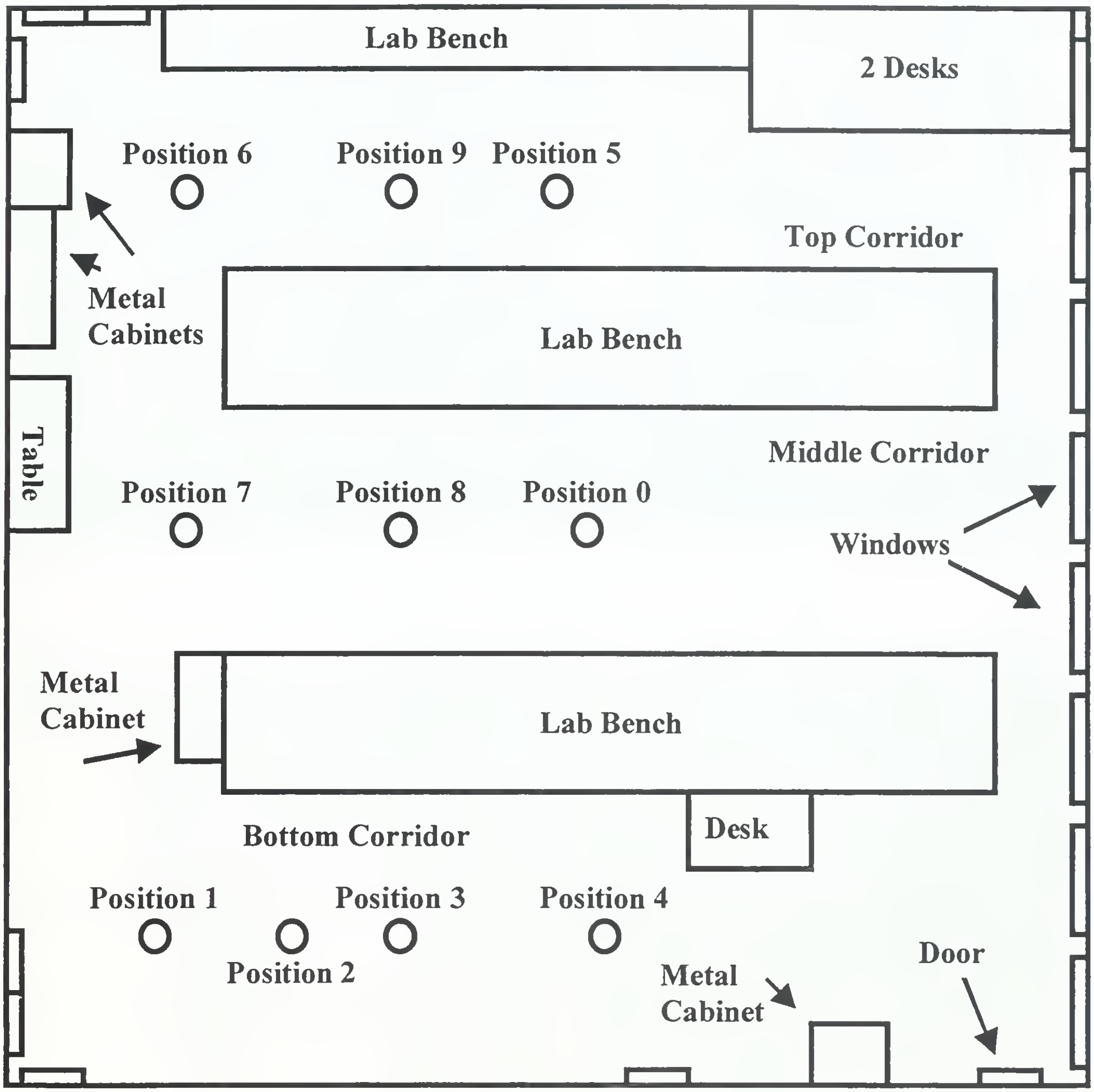


Figure 16. Simple illustration of test environment for single and multiple robot trials showing starting positions and significant geographical features.

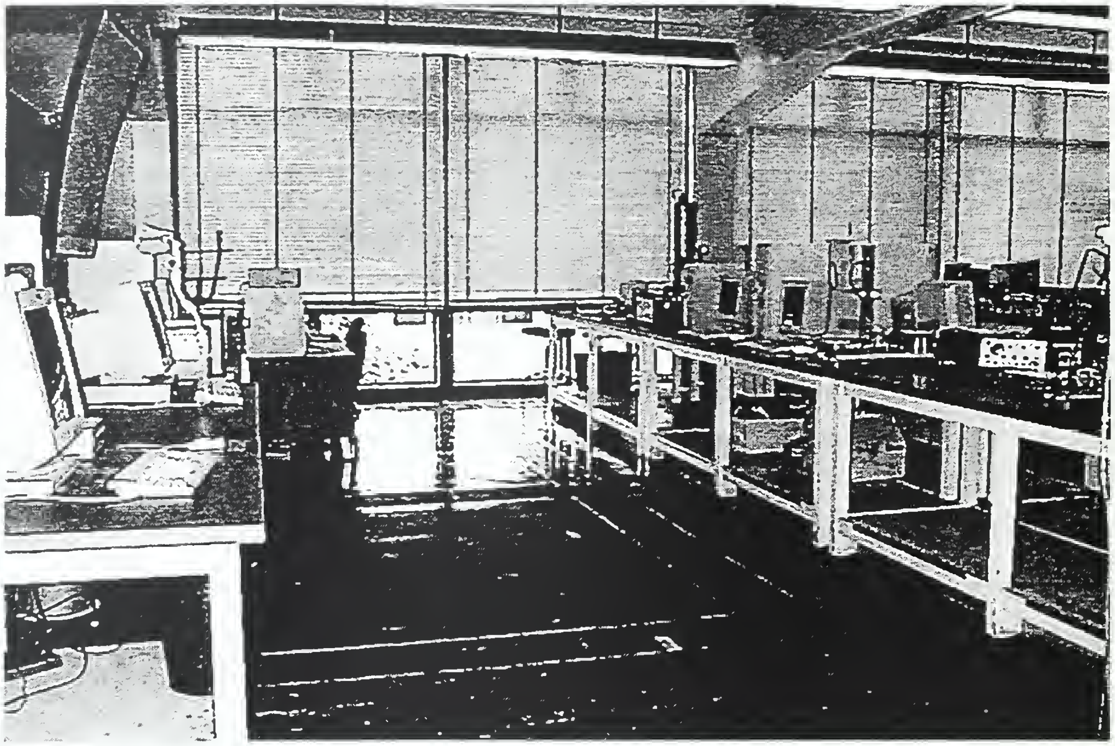


Figure 17. Top corridor of test environment.

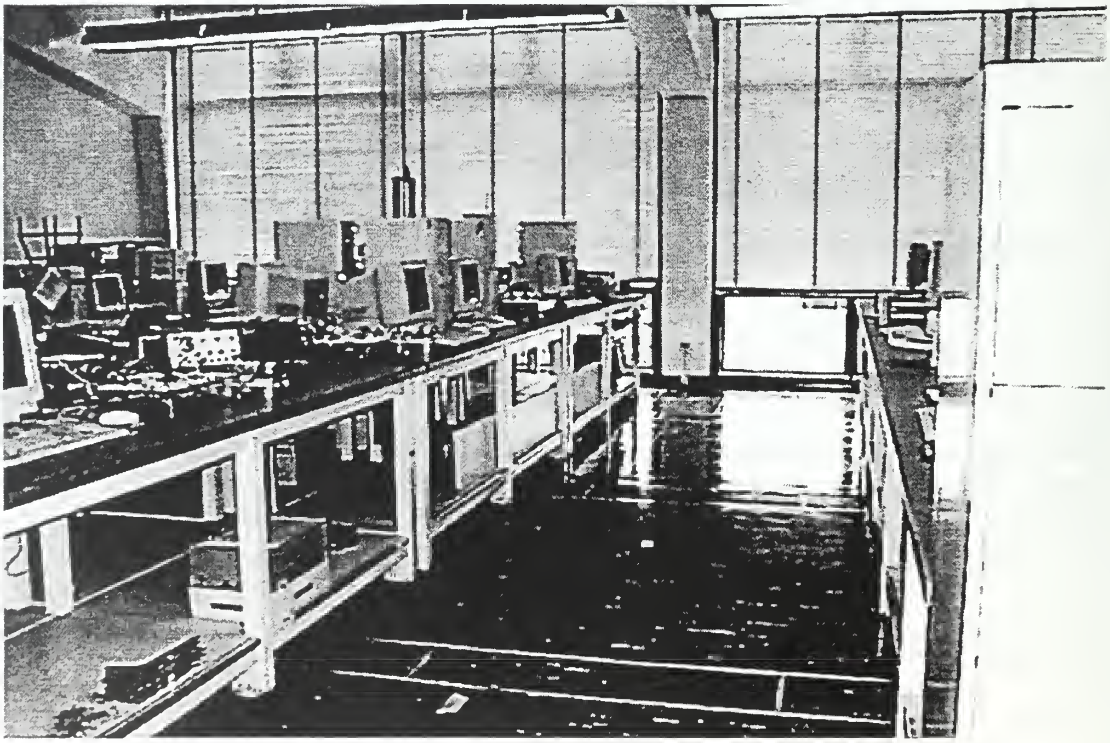


Figure 18. Middle corridor of test environment.

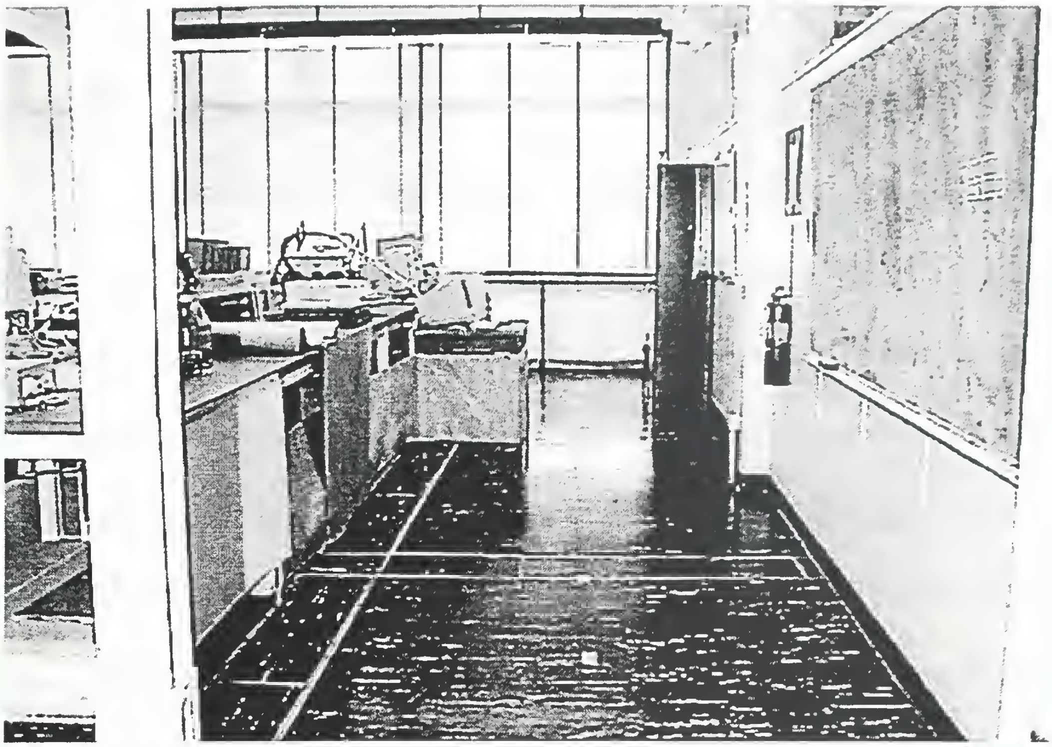


Figure 19. Bottom corridor of test environment.

2. Experimental Variables

After the basic robotic exploration and mapping system was functional it was decided to concentrate on examining a few easily-manipulated variables in order to attempt to optimize the system for the given test area.

a. Given Area to be Mapped

The first thought was to minimize the area the robot would be expected to map. This variable is set in the file *grid.h* and sets the size of the room the robot will map. Minimizing this would seem to ensure the finest detail possible for the given evidence grid resolution (as described below). It proved not to be practical to set it exactly to the actual size of the test area. In a perfect world the robot could have been

instructed to map a 37 by 37 foot room and all would be well. However, as odometry errors began to accumulate during a test run, the robot became confused near the edges of the room when its now-inaccurate odometric encoders indicated that it was outside the boundaries of the expected area.

In addition, specular reflection errors off of objects near the boundaries (especially the windows) caused false sonar returns from outside the boundaries set for the room. This caused additional errors. To alleviate these difficulties a 3.5 foot safety margin was added on each side of the room boundaries, resulting in a 44 by 44 foot area that the robot expected to map. This reduced the overall resolution slightly, but resulted in more consistently successful mapping efforts.

b. Evidence Grid Resolution

Also in the file *grid.h* the evidence grid resolution is set. In order for the evidence grid method to correctly fuse two different grids the grids must be symmetrical and a power of two. Varying resolutions from 64 by 64 cells to 512 by 512 cells were tested.

The initial testing with a setting of 512 by 512 cells resulted in very noisy sonar data and many small frontiers. These small frontiers were often found to be grouped around one large object, especially one with many projections such as a chair or table. It was hoped that setting a coarser resolution would result in quicker mapping of large areas and less noise from the arms or legs of chairs and tables. It soon became

evident that using a very coarse resolution, such as 64 by 64 cells, did result in a shorter mapping trial, but not for the desired reasons.

With the room size mentioned above of 44 by 44 feet and using 64 by 64 cells in the evidence grid, each cell was about 68 in^2 or 8.25 inches on a side. This is a rather large size for a cell compared to the size of objects in the test environment. As expected, the noisy sonar returns were blurred into fewer cells, but the unfortunate side effect was that now large cells that were only partially filled were marked as completely filled, whereas with finer detail these areas would have been resolved into open space. With the coarse detail setting the robot soon marked all possible paths as blocked by obstacles when in fact there was still many open paths for it to travel. This is illustrated in Figure 20. Here the resolution was set at 64 by 64 cells and the robot was started at position zero in the center of the room. Even though the corridor was open, noisy returns were still blurred together to the point where the robot determined that is was completely blocked.

Numerous trial-and-error investigations led to the choice of 256 by 256 cells for the grid resolution in conjunction with the "trustworthy" sonar range discussed below. Again using the room size of 44 by 44 feet, but now with 65536 cells, each cell in the evidence grid was approximately 4.25 in^2 or less than 2.1 inches on a side. This was the best compromise found between reducing noisy data and having fine enough detail to navigate the robot and map properly. It is important to note that these results were specific to the environment the mapping tests were conducted in and will most likely be quite different in dissimilar circumstances.

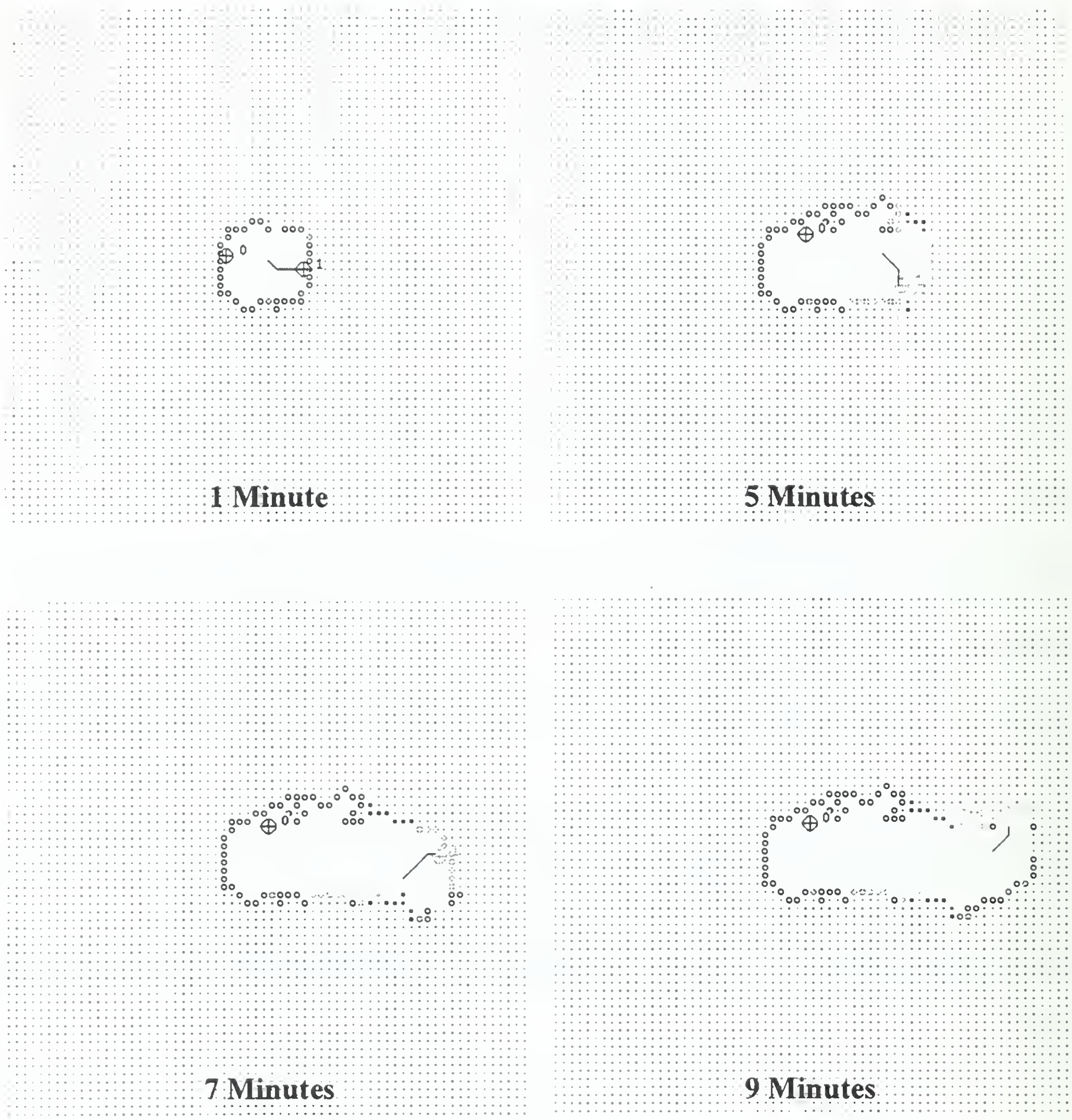


Figure 20. Illustration of robot exploring corridor using coarse discrimination (64 by 64 cells). Large individual cell size causes false determination that ends of corridor are blocked.

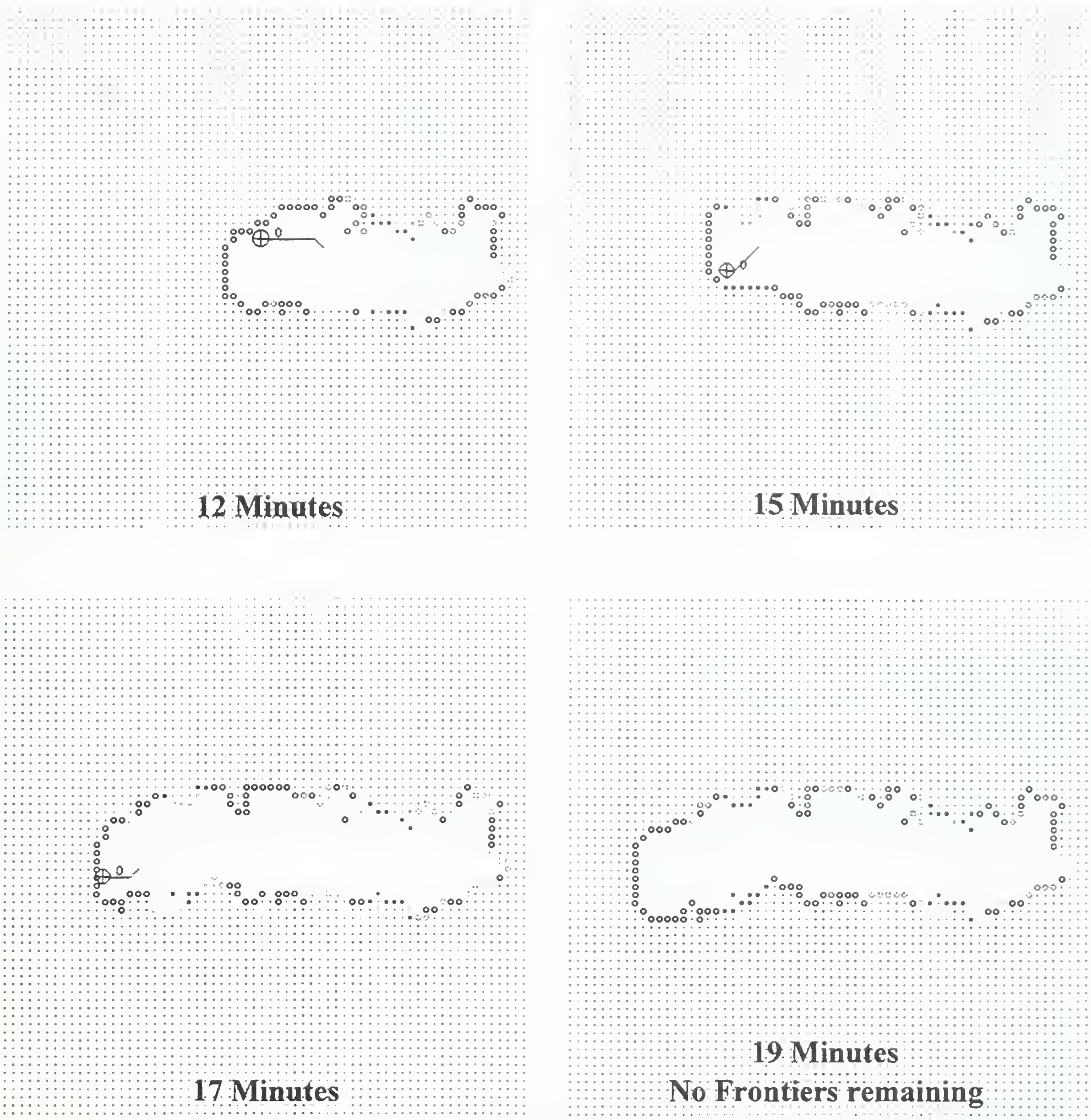


Figure 20 continued. Illustration of robot exploring corridor using coarse discrimination (64 by 64 cells). Large individual cell size causes false determination that ends of corridor are blocked.

c. *"Trustworthy" Sonar Range*

The final option that was manipulated in the *grid.h* file was the *MAX_SONAR_RANGE* variable. This variable sets the "trusted" range for sonar sensor readings. Readings indicating distances further away than this setting will be disregarded for the purposes of map building and exploration. As was mentioned earlier, setting this to a lower value than the 10 foot range used with the NOMAD 200 seemed to be the best way to reduce the problem of specular reflection. Also, as mentioned previously, there was a penalty in setting this too low in the increased amount of travel the robot would be required to do and the subsequent increase in localization error.

After many trials it was found that a six foot range was adequate to reduce many (but not all) specular reflection effects and did not seem to compromise the robot's localization capability to any great degree. However, if an additional localization method is added to the NOMAD SCOUT platform in the future it is recommended that this range be further reduced in order to further mitigate specular reflection problems. Figure 21 shows a sequence of maps created during a NOMAD SCOUT mapping sequence with the *MAX_SONAR_RANGE* set to 10 feet and a grid resolution of 256 by 256 cells. The robot was started at position zero in the center of the test area. Note the numerous and extensive specular reflection effects from the areas near the benches and windows. These false returns created numerous small, false frontiers that the robot attempted to explore, but could not reach.

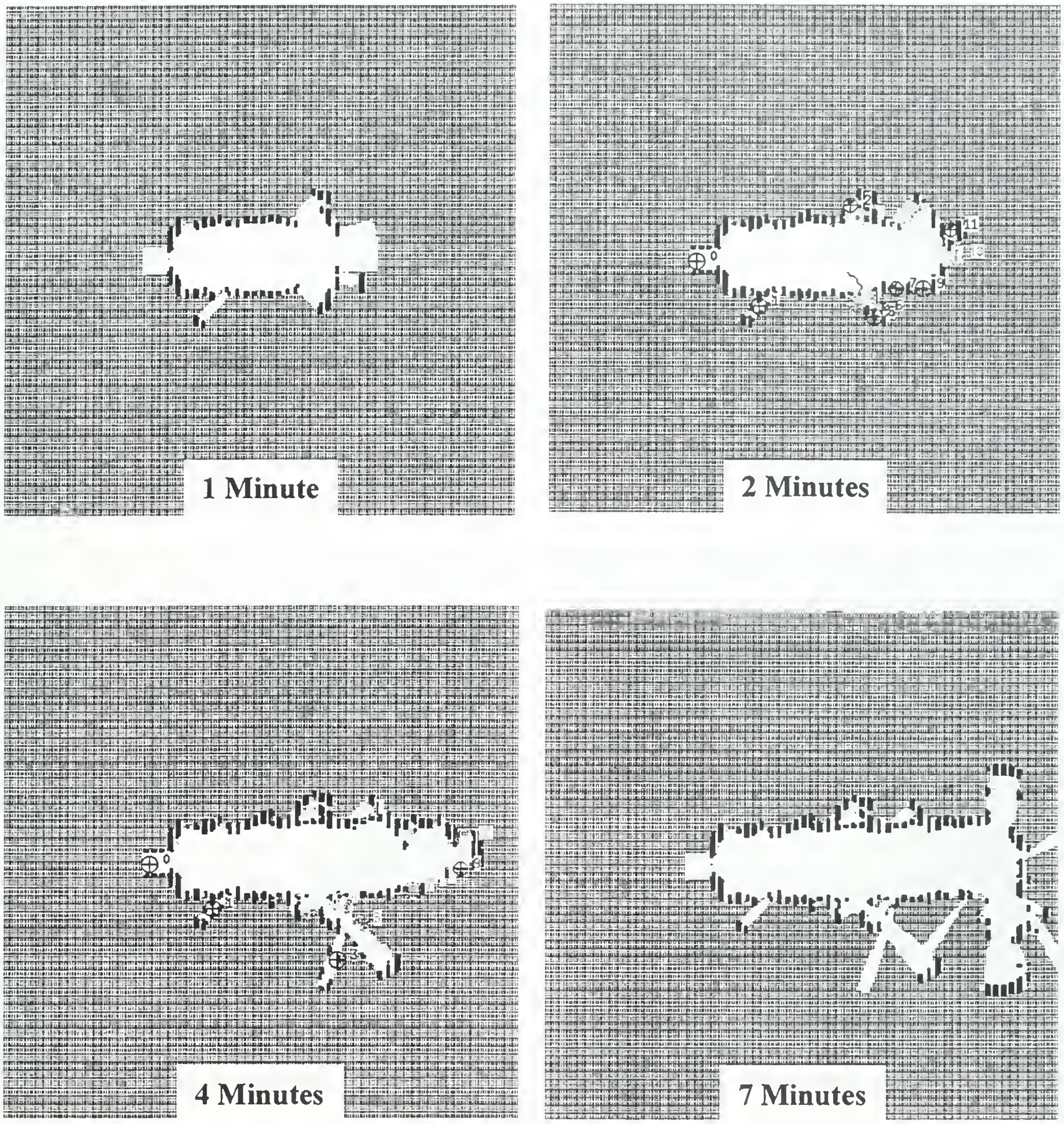


Figure 21. Illustration of false sonar returns and subsequent poor mapping results due to extensive specular reflection when using 10 foot sonar range.

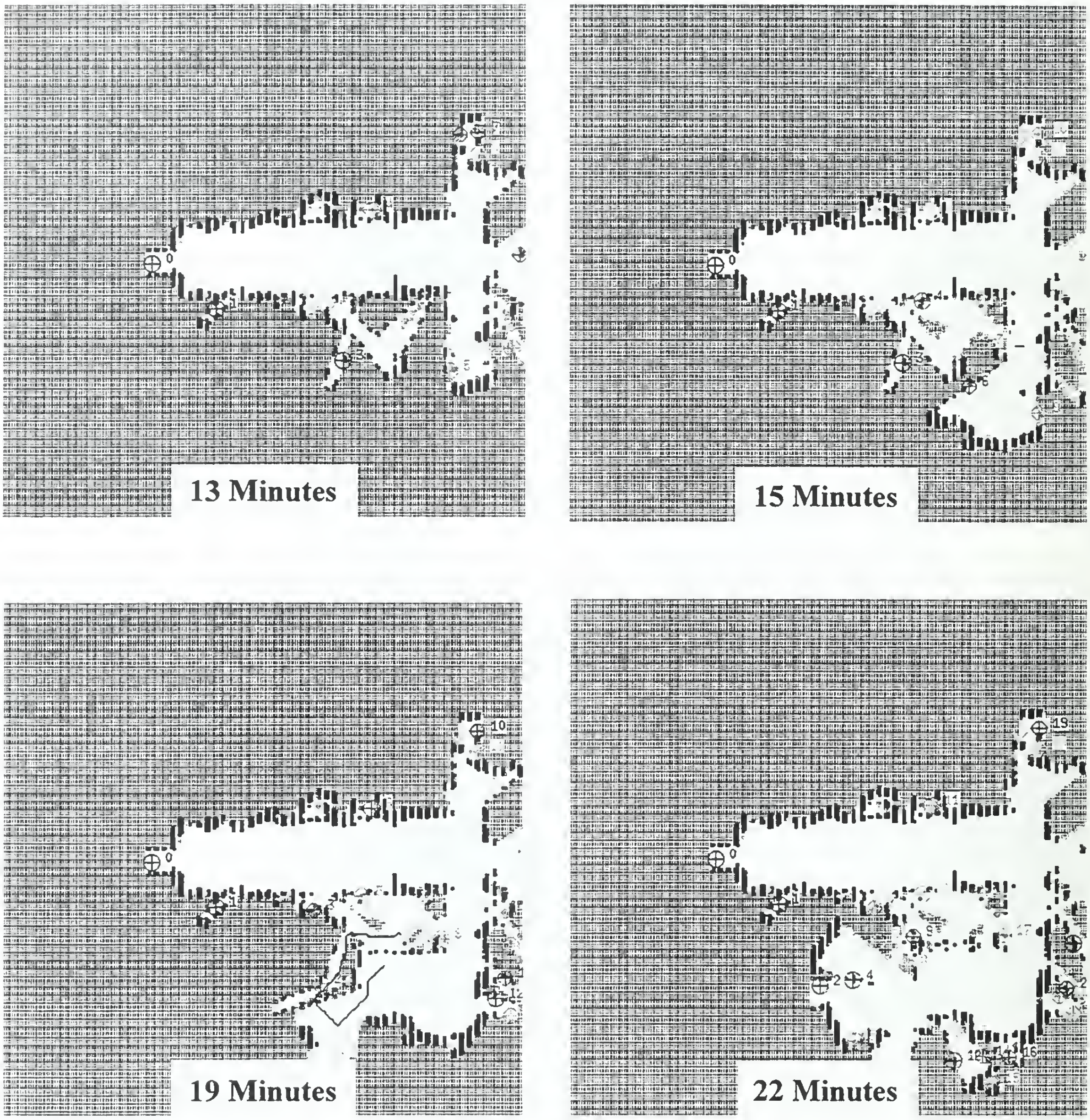


Figure 21 continued. Illustration of false sonar returns and subsequent poor mapping results due to extensive specular reflection when using 10 foot sonar range.

3. Trial Runs and Results

One thing is immediately noticeable from the many trial runs conducted with one robot in the initial research: as currently implemented no single robot alone will be able to map the entire test area. On average, after 20-25 minutes of travel the odometry errors

become so large that further mapping efforts are actually counterproductive. Continued mapping at that point, with localization so badly compromised, will most likely begin to overwrite previously accurate areas of the evidence grid map with inaccurate data. This was seen many times in longer trials. In the current implementation there is not enough time before odometry error becomes fatal to the exploration and mapping effort for the robot to cover the entire space. Thus the need for multiple robot exploration and mapping is evident.

Figure 22 illustrates a typical trial run with the "standard" settings of a 256 by 256 cell evidence grid resolution and a maximum trusted sonar range of six feet. This run was started from position zero in the center of the test area. Mapping efforts continued well for about the first 15 minutes. After that time, localization errors began to interfere with the robot's ability to navigate to new frontiers. Localization continued to get steadily worse, especially rotational tracking. By the 21^{st} minute of the experiment the robot was actually travelling in the opposite direction than it indicated that it was moving. This seemed to be a common trend among many of the trials. The small movements that the robot makes during sonar sensor sweeps at new frontiers as well as the many small turning motions the robot makes as it travels seem to affect the rotational localization much more quickly and much more detrimentally than the translational localization.

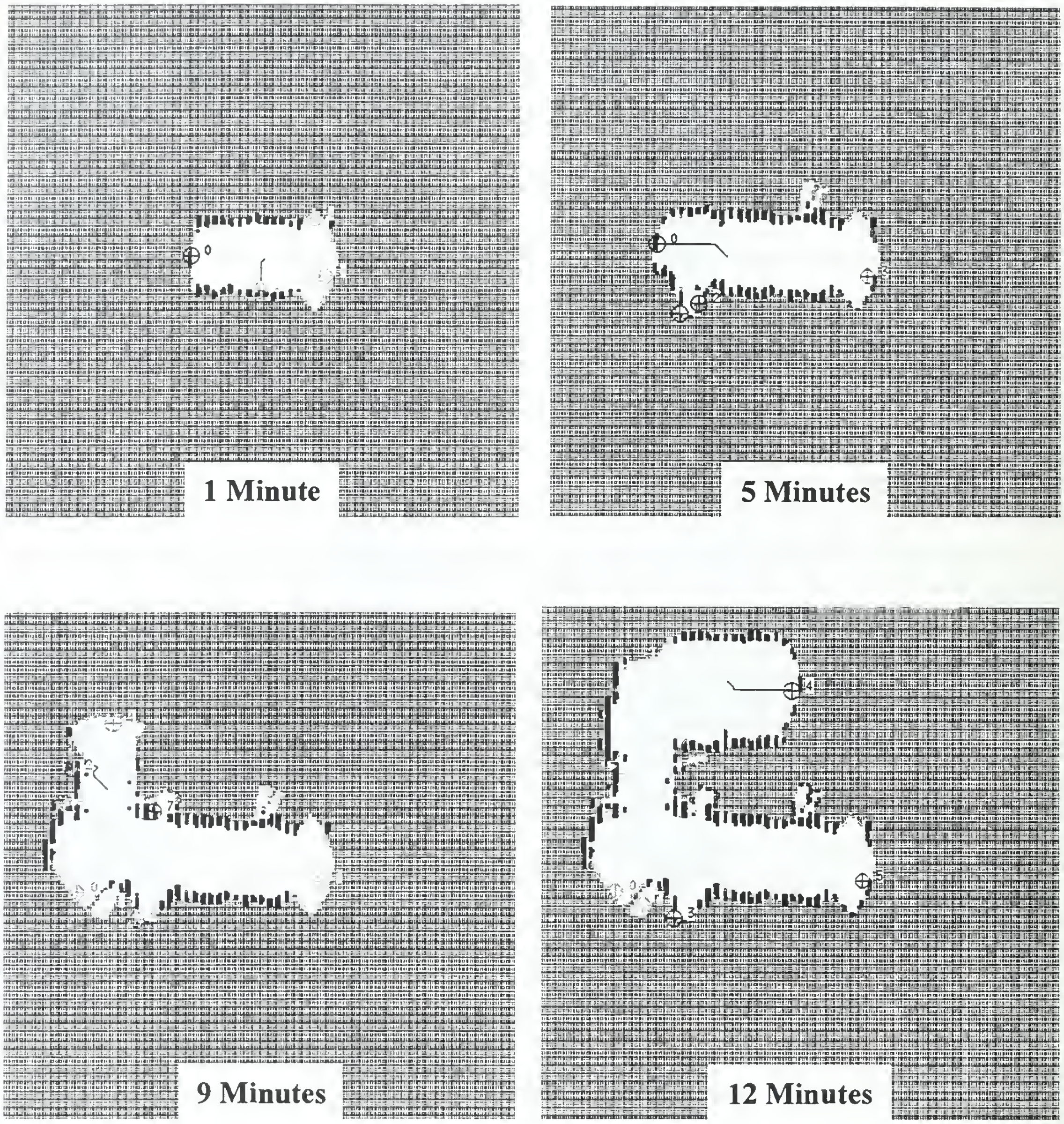


Figure 22. Illustration of fatal localization error beginning about 15 minutes into the mapping and exploration phase.

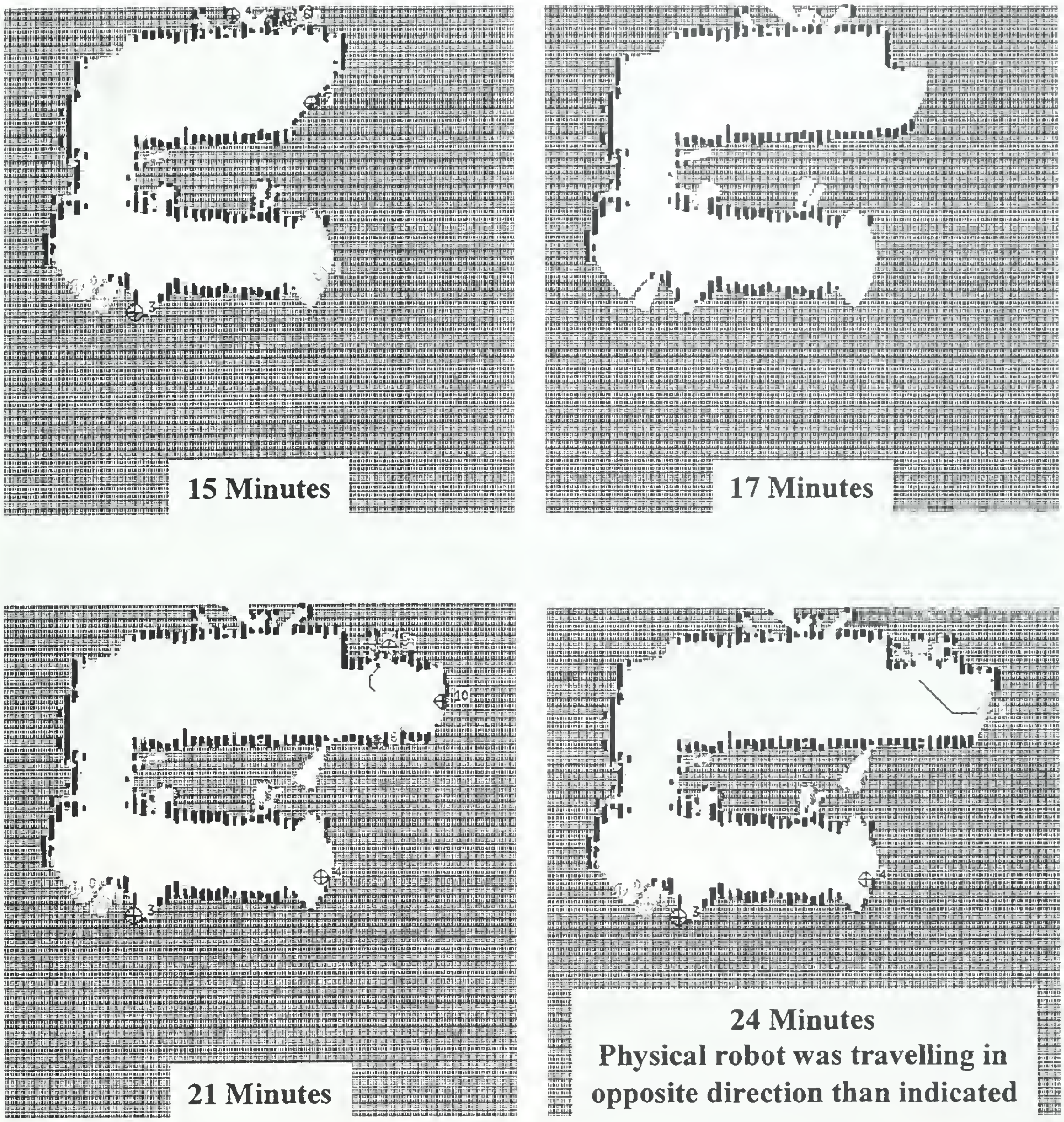


Figure 22 continued. Illustration of fatal localization error beginning about 15 minutes into the mapping and exploration phase.

However, other trials showed that localization errors could also cancel each other out and allow longer periods of mapping. These maps will be distorted compared to the actual "ground truth" of the area mapped, but they will still have recognizable, albeit distorted, geographical features such as corridors, corners, etc. Figure 23 is an example of such a trial. This run was conducted under the standard conditions with the robot

initially started at position one. Between the nine and 19 minute point in the trial the robot was stuck in a small area between the two benches trying to explore many small, inaccessible frontiers. By the time it "broke free" its odometry was obviously distorted, but it was possible to still recognize map features produced for another 12-14 minutes.

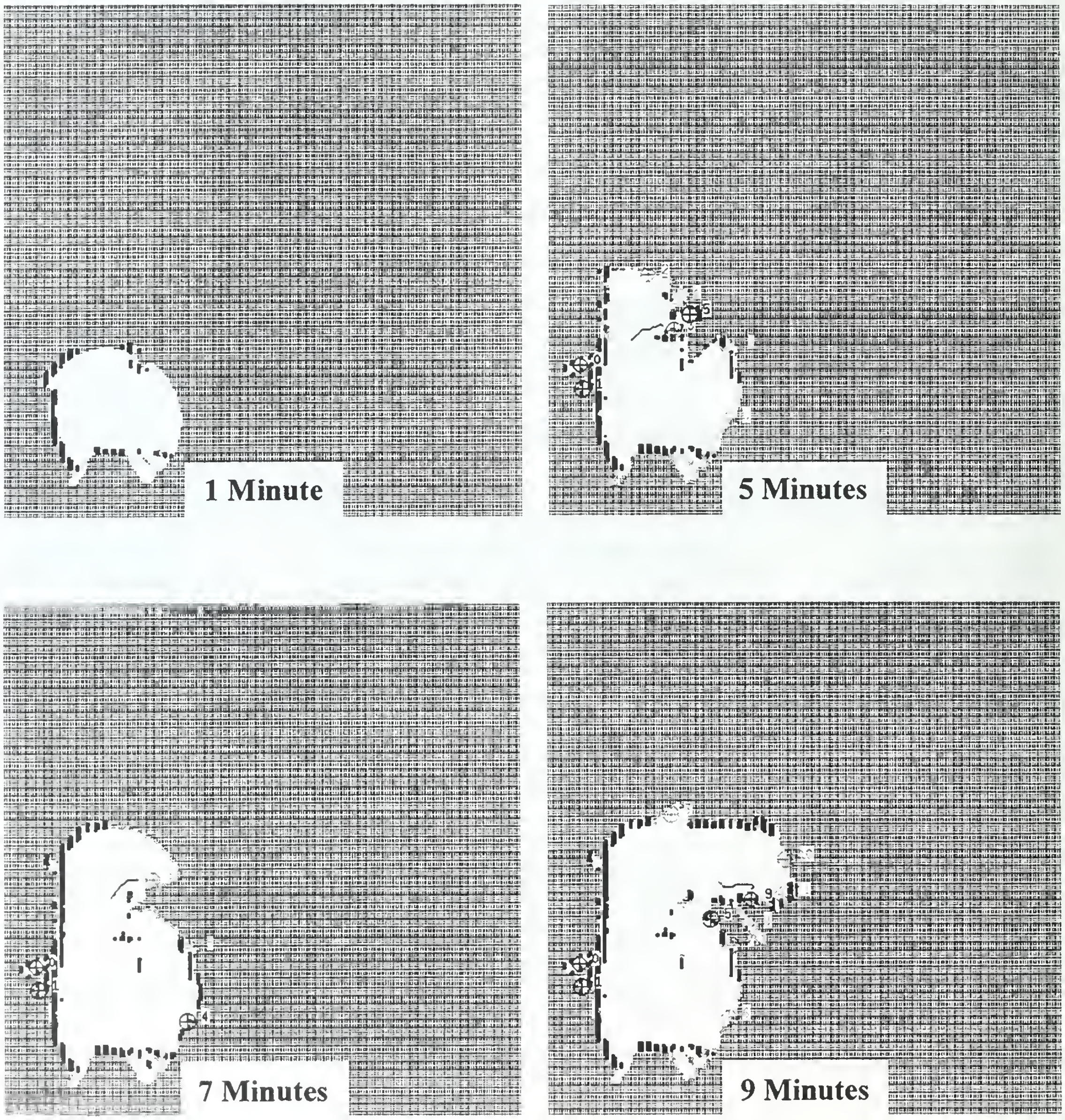


Figure 23. Illustration of robot getting temporarily trapped in a small area, breaking free, and then continuing to produce a recognizable, although distorted, map for several minutes longer.

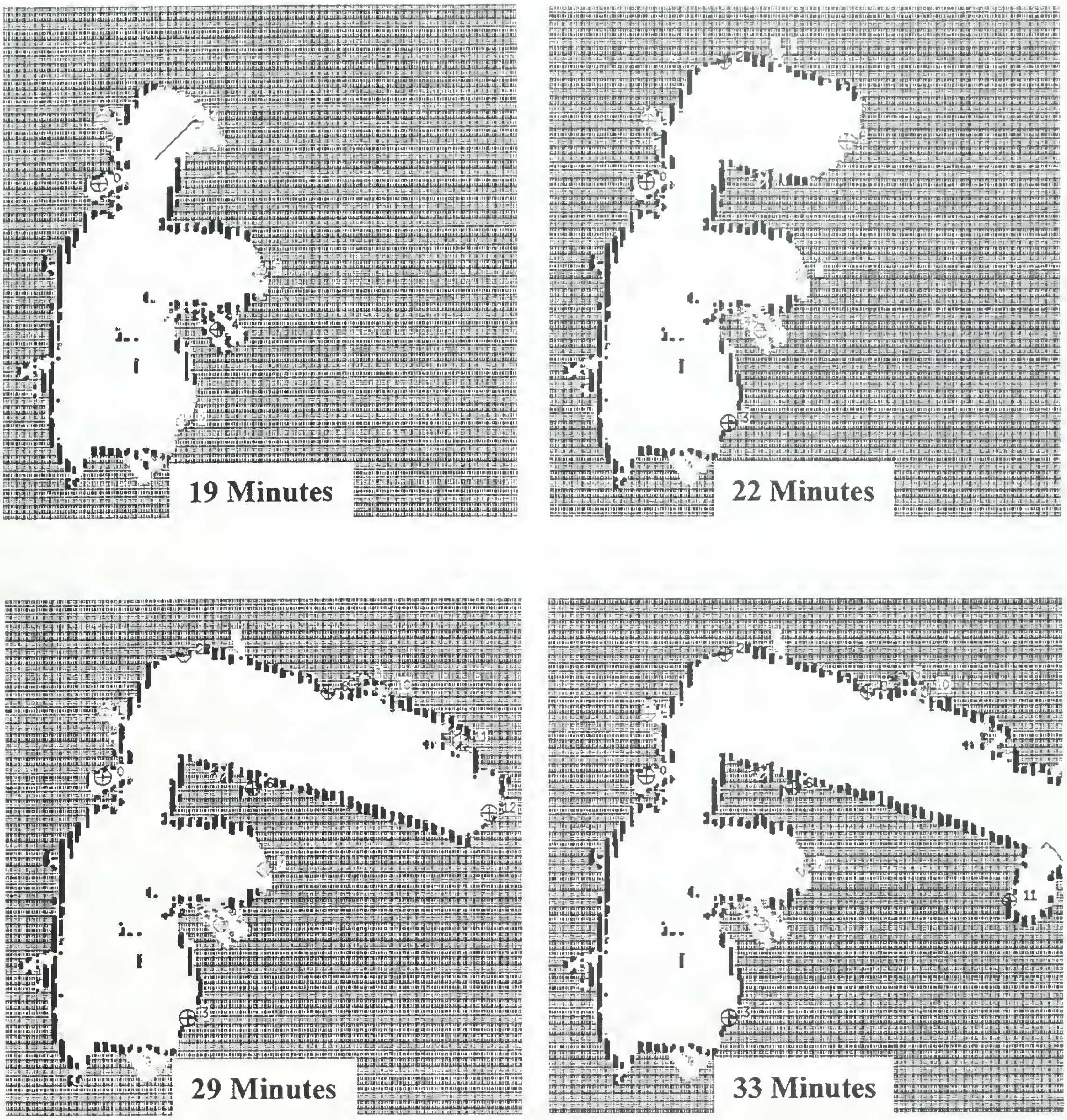


Figure 23 continued. Illustration of robot getting temporarily trapped in a small area, breaking free, and then continuing to produce a recognizable, although distorted, map for several minutes longer.

All single robot trials continued to point toward the need for multiple robots acting simultaneously in order to map the test area. Under the current implementation a single robot cannot map the area before localization errors render it incapable of further exploration and mapping.

B. MULTIPLE ROBOT MAPPING EFFORT

Multiple robot mapping efforts provide a number of mixed results. In some circumstances the use of multiple robots dramatically decreases the amount of time required to map a given area compared to a single robot mapping the same area. In fact in some cases multiple robots were able to map an area that the single robot could not complete due to buildup in localization errors. However, under other circumstances multiple robot mapping can be less efficient than expected and actually worse than single robot efforts. There also appear to be some issues with the effects of controlling many robots simultaneously on network performance and reliability.

1. Multiple Robot Test Conditions

The test area for all multiple robot trials was the same as for the single robot trials, the 37 by 37 foot research room described previously. All robot processes, as well as the *Nserver* program, were run on the same Sparc 20 workstation used for the single robot trial and the robot processes controlled their respective robots via the same wireless Ethernet connection. Using the same workstation to run all the robot processes simplified the sharing of map data between the client and server processes, but did lead to an overall slowdown in the speed at which the individual processes ran as more robot processes were added. For all the multiple robot trials each individual robot was similarly configured with an evidence grid resolution of 256 by 256 cells and a trusted sonar range of six feet.

2. Trial Runs and Results

One major finding from the multiple robot trials was that there was not quite the consistency or quality of performance improvement that had been expected. In a perfect implementation if a single robot can map *X* area in *Y* time, then *N* robots should be able to map *X* area in *Y/N* time (or conversely map *N*X* area in *Y* time). While such extreme levels of performance improvement were not expected at this point in the research, it was expected that there would be somewhat more improvement and more consistency in improvement than was seen. This will be discussed further below.

3. Beneficial Effects

Some multiple robot trial runs did show significant improvement in mapping efforts over single robot trials. This was especially true when the robots started in widely varying geographical positions in the test environment with well-known initial starting coordinates. In these cases each robot mapped its local area in the same manner as in the single robot trials and the server robot process consolidated the map data. Figure 24 illustrates this process for an average two-robot trial.

In the trial shown in Figure 24 one robot was started at position zero and the other robot was started at position five. With the two robots separated by an obstacle (in this case one of the benches) they explored their general area without interference from each other. One important thing to note about this trial is that the robot in the middle corridor suffered networking problems after approximately 20 minutes. It stopped

mapping, but the robot in the top corridor continued mapping. This illustrates the benefits of a distributed system without reliance on a central controller to operate.

As the still functioning robot continued to map the top corridor localization errors soon began to degrade its navigation capabilities after about 23 minutes. At the 30 minute point in the trial further mapping efforts are futile. Note the specular reflection effects on the robot in the middle corridor near the windows and along the benches in the middle corridor.

Figure 25 illustrates a representative three-robot trial with the robots starting in widely separated positions. In this case the robot were started at positions two, eight, and nine. Again, each robot began to explore its local area without interference and the server robot process collected the local sensor data from each client robot process, fused the data, and distributed a new global map to all the robots in the system.

For the 20 minutes that this experiment ran localization for each individual robot was maintained relatively well. The combination of the three robots managed to explore and accurately map more of the test area than a single robot would have been able to in the same amount of time. More importantly, even if a single robot could manage to navigate through the same amount of area that the three robots covered, its mapping accuracy would be much worse than the three-robot system. The longer time required for a single robot to cover the same area as three robots would lead to many more localization errors on the single robot compared to those of any individual robot in the three-robot system.

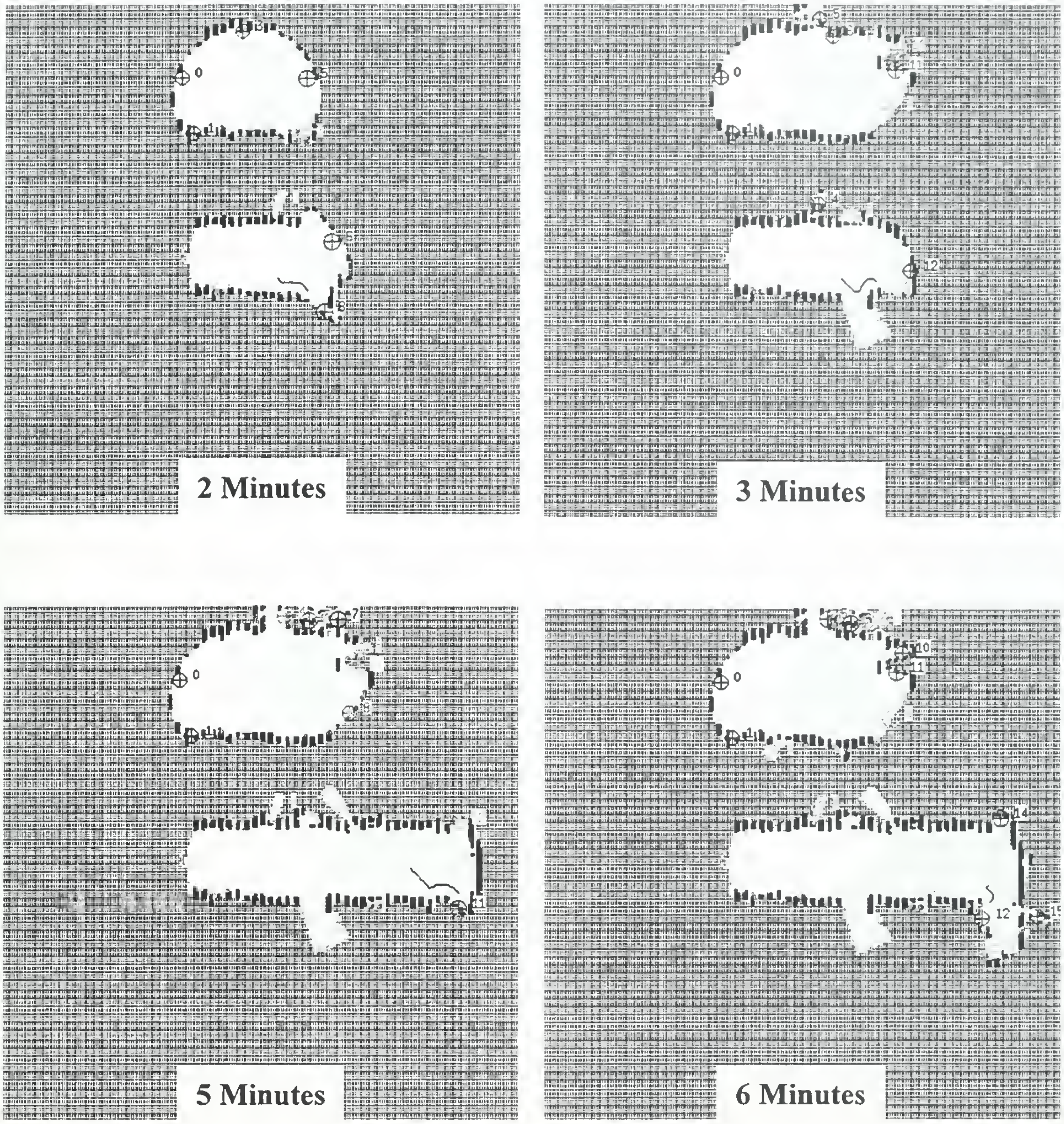


Figure 24. Illustration of a two-robot exploration trial with the robots starting in widely different positions.

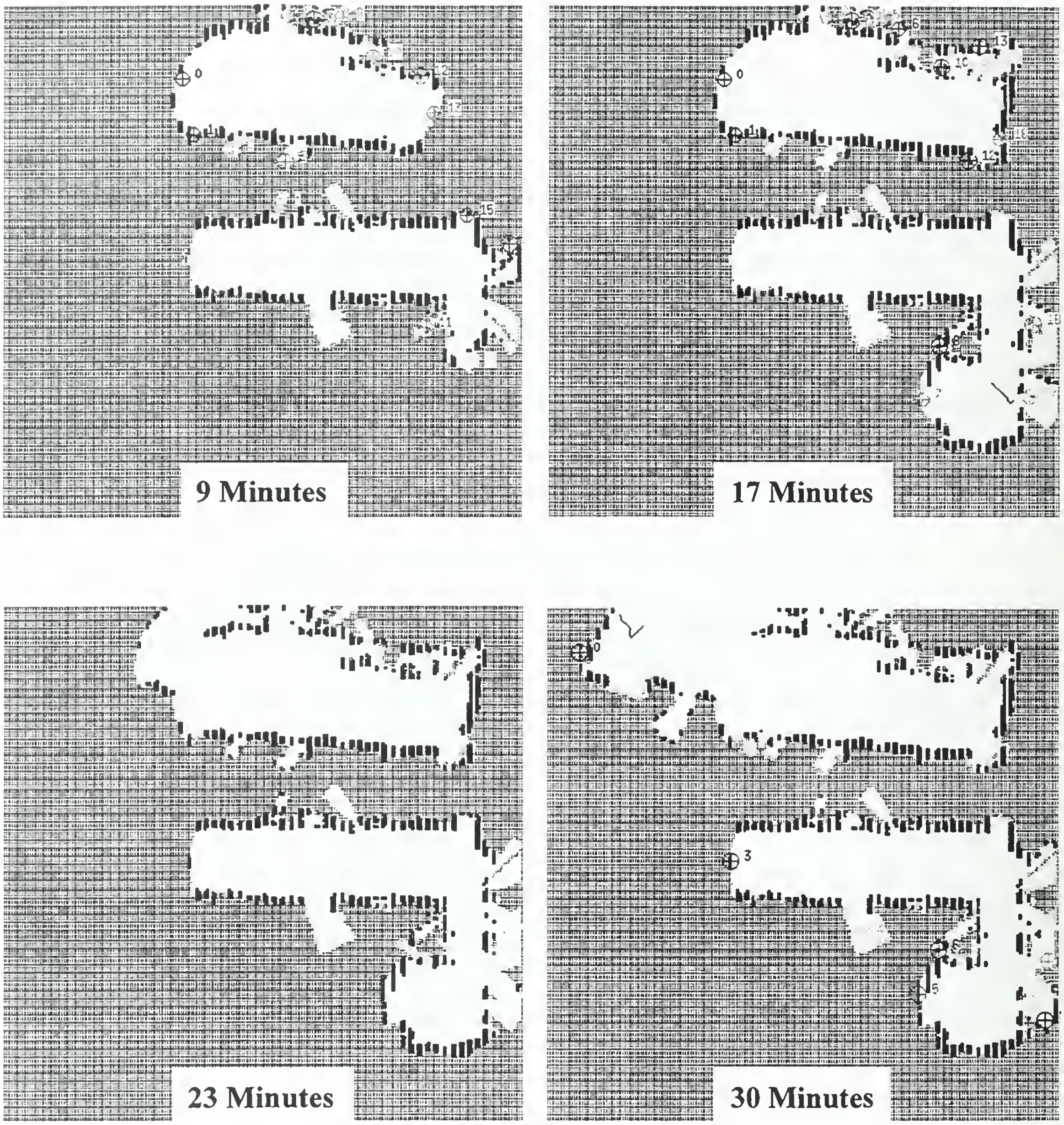


Figure 24 continued. Illustration of a two-robot exploration trial with the robots starting in widely different positions.

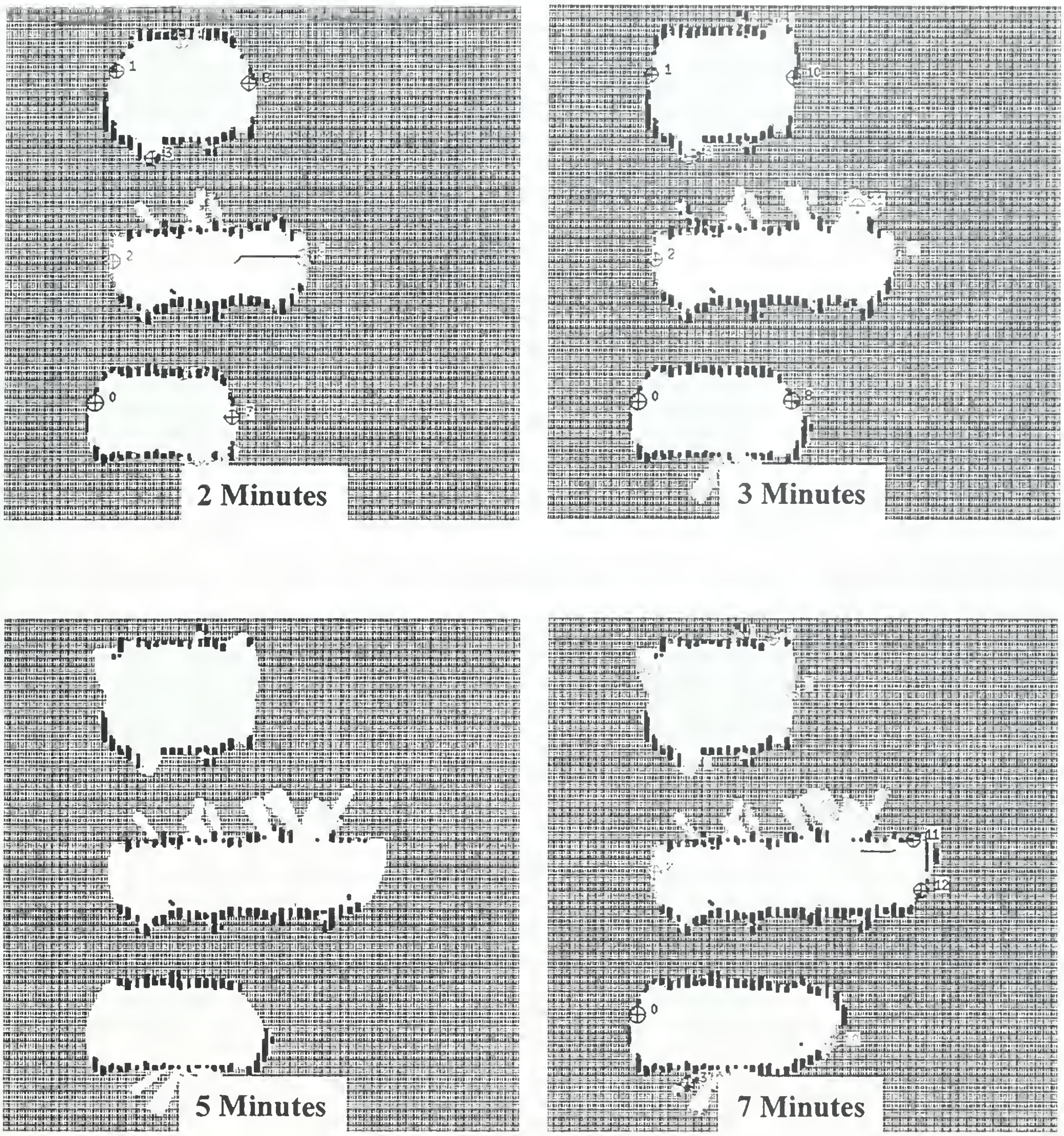


Figure 25. Illustration of a three-robot exploration trial with the robots starting in widely different positions.

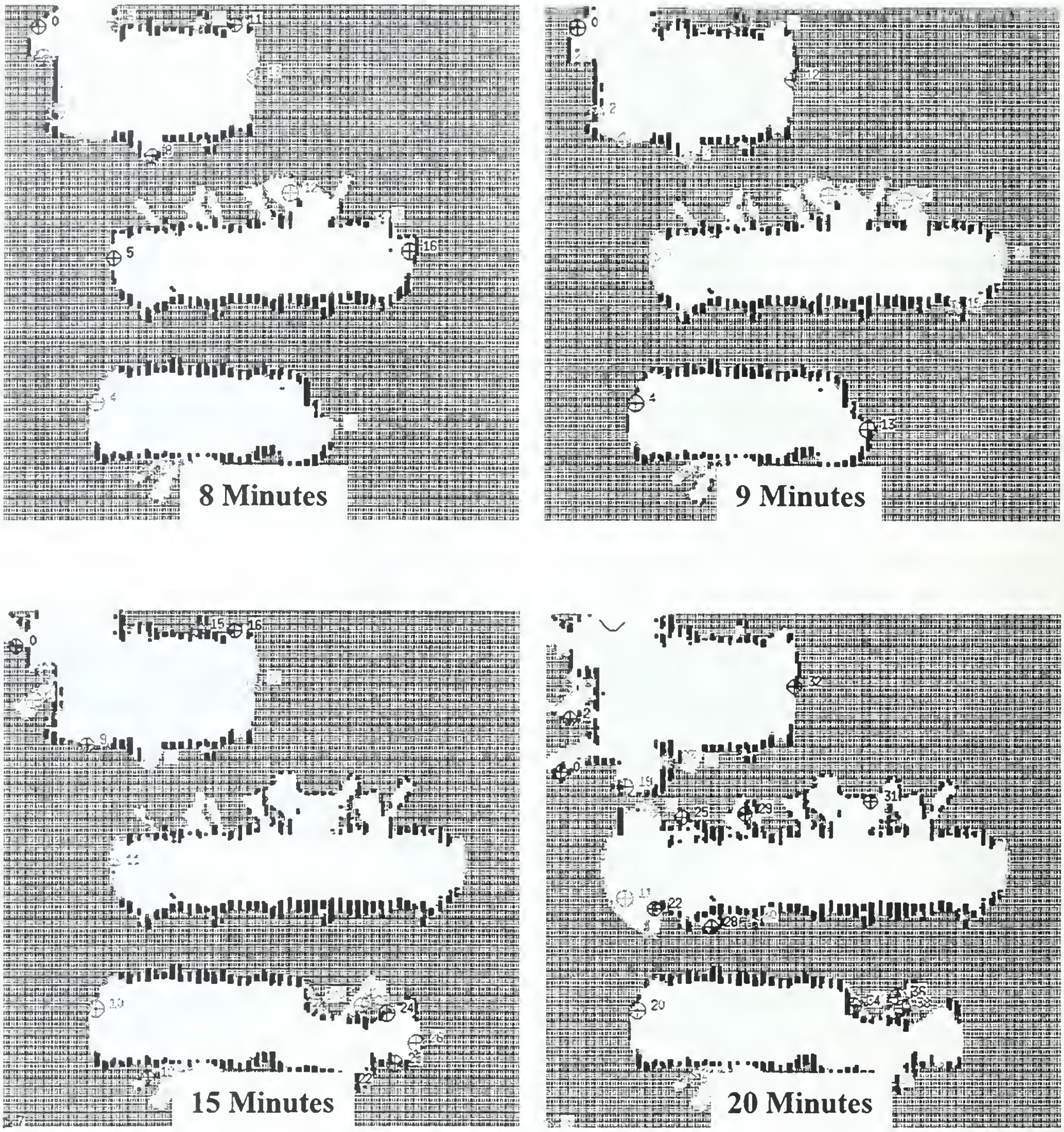


Figure 25 continued. Illustration of a three-robot exploration trial with the robots starting in widely different positions.

4. Counterproductive Effects

Not all the results of using multiple robots were beneficial in terms of mapping accuracy and efficiency. There were several sets of circumstances that often led to multiple robot trials being less efficient than it would be assumed they would be and in

some cases even less efficient than a single robot system. The most common cause of multiple robot inefficiencies was near proximity of one robot to another robot during the exploration and mapping process.

a. "Follow The Leader" Behavior

Two related multiple robot exploration problems came to be known as the "Follow The Leader" and "Dancing Robots" behaviors. Both of these behaviors happen when two or more robots are near (within the trusted sonar range) one another. The "Follow The Leader" behavior can be described as one robot appearing to follow another robot through the test environment and apparently mapping the same area that the leader robot has just mapped. Numerous real-life and simulation trials have revealed two predominant causative factors for this behavior.

The primary cause seems to be the time delay in map data being passed from one robot to another and when that data is processed during the exploration routine. When two robots are near one another they will usually see the same frontiers nearby. Commonly one robot will proceed to a nearby frontier and the other robot will proceed to another nearby frontier that is slightly closer to it. However, if the second robot finishes its exploration of the first frontier and still has not received the map data from the first robot's exploration there is a good chance that it will travel to the same frontier that the first robot just explored.

The second robot will probably not receive and process the first robot's map data until after the second robot has already mapped the same area that the first

robot just left. By then the first robot has moved on to a nearby frontier, which is now also the next frontier that the second robot will attempt to explore. This process can continue with the second robot always one set of map data behind the first robot and following it all over the test area.

The other common cause of this behavior is that the second robot senses the first robot that is nearby as an obstacle in the environment with unexplored frontiers around it. While the second robot heads for the first robot's position the second robot moves away to explore a nearby frontier. Once the second robot reaches the first (leader) robot's previous position it makes a sensor sweep, again notes the nearby first robot as an obstacle with new frontiers to explore around it, and again the "Follow The Leader" process continues. In the best case this type of behavior merely reduces the effectiveness of the system by one robot (the following robot). In the worse case, instead of the "Follow The Leader" behavior, the "Dancing Robots" behavior occurs and both robots are rendered ineffective.

b. "Dancing Robots" Behavior

The "Dancing Robots" behavior can be described as two or more robots circling around or moving back and forth near one another for extended period of time and remaining in a relatively small area of the test environment. One variant on this behavior is vaguely reminiscent of the "do-si-do" movement, typically seen in square dancing, where two robots will swap places as if they are swinging each other around. As

interesting as this behavior is to observe, it does not aid in the exploration and mapping process.

This behavior is also caused by a robot sensing another robot as an obstacle with new frontiers around it to be explored. However, in this case both robots sense one another instead of just a follower sensing a leader. Whenever one of them moves to explore the frontiers around the other, the other does the same and thus the robots "dance" around one another. This process can continue indefinitely until one robot is stopped or another nearby frontier that is not caused by a mobile robot is chosen for exploration. Besides keeping two robots from exploring the rest of the area, the constant "dancing" motions have an extremely detrimental effect on localization.

Figure 26 is an example of this inadvertently happening in a three-robot trial. The robots were initially started at positions one, six and seven. Exploration continued normally for the first few minutes. After about 5-7 minutes the robots in the two lower corridors came close enough together to sense one another. At that point many small frontiers were generated by each robot during its exploration process as the other robot moved through the area while the sonar sensor sweep was taking place. The robots began to "dance" around one another for the next 6-7 minutes until one of them was halted. At that point the other robot was able to "break free" because no new frontiers were being generated by a moving robot. However, by this point the robot's localization has been compromised due to the effects of small movements around the other robot.

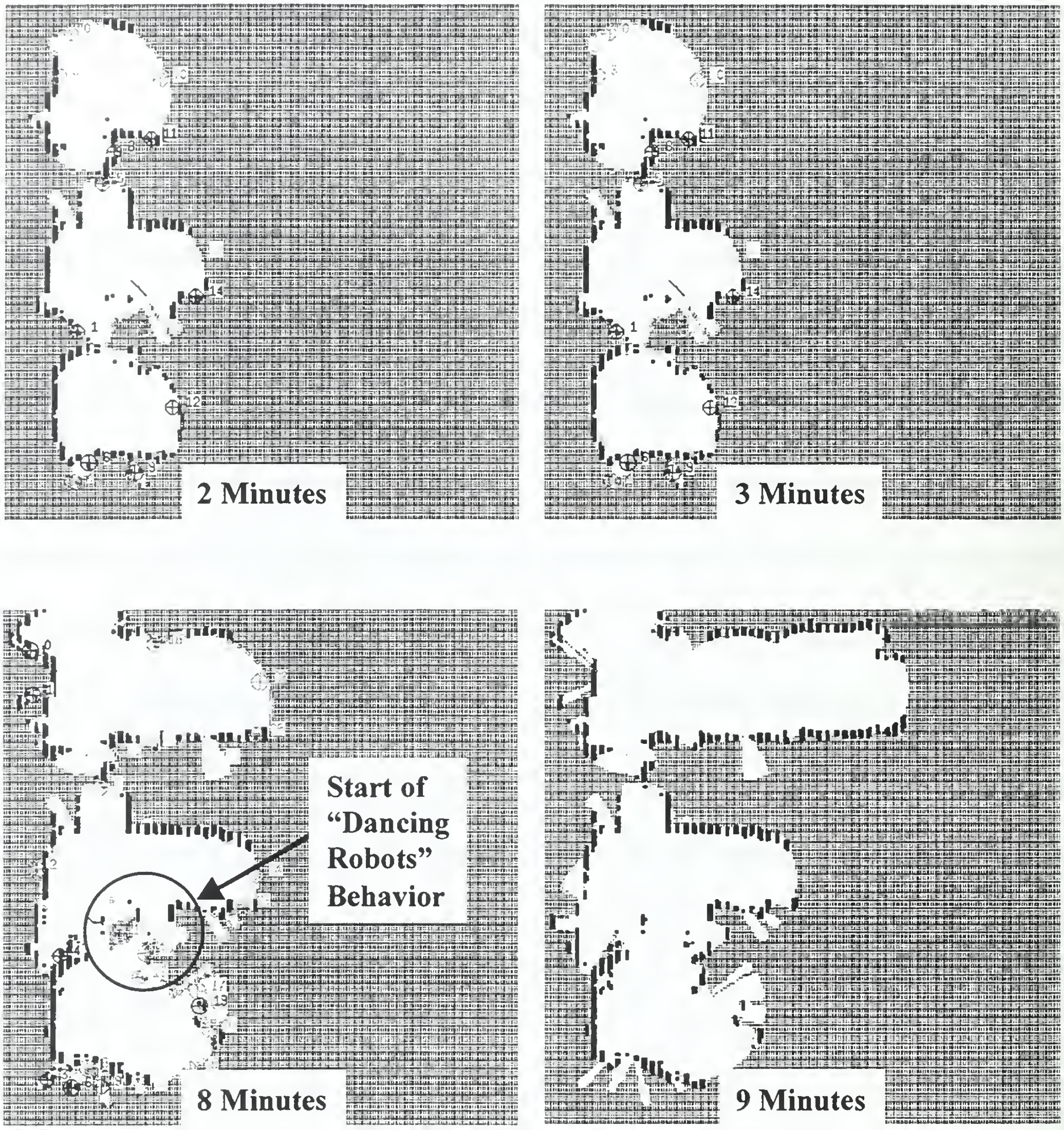


Figure 26. Illustration of "Dancing Robots" behavior during a three-robot trial.

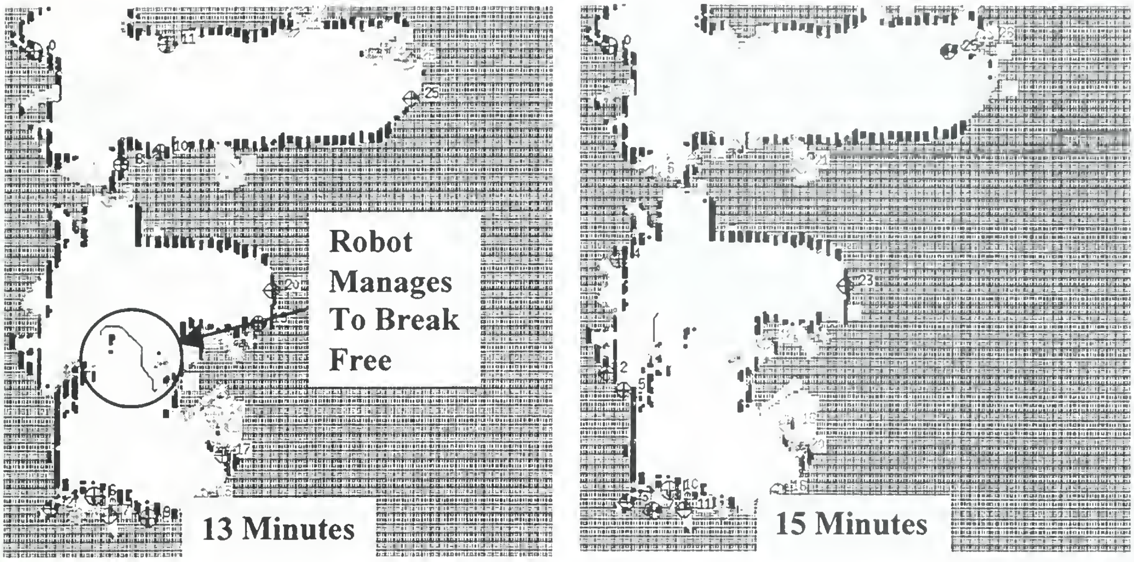


Figure 26 continued. Illustration of "Dancing Robots" behavior during a three-robot trial.

Figure 27 is a trial done with all the robots started in a very near proximity to one another. The initial starting points were positions one, two, and three. *Nserver* can be used to display the robots relative positions in the test environment based on the encoder data sent from the robot back to its respective controlling process. This option was used to track the robots' positions in the test area. The robots are labeled on the figure for ease of reference.

At the start the first robot is slightly farther away from the other robots and does not see any frontiers generated by detecting the other robots as obstacle. In comparison, the second and third robots are much closer together and generate many small frontiers as they detect each other nearby. At first it appeared that the second robot might just follow the third robot, but the third robot detected no frontiers closer than those around the second robot and thus began the "dance."

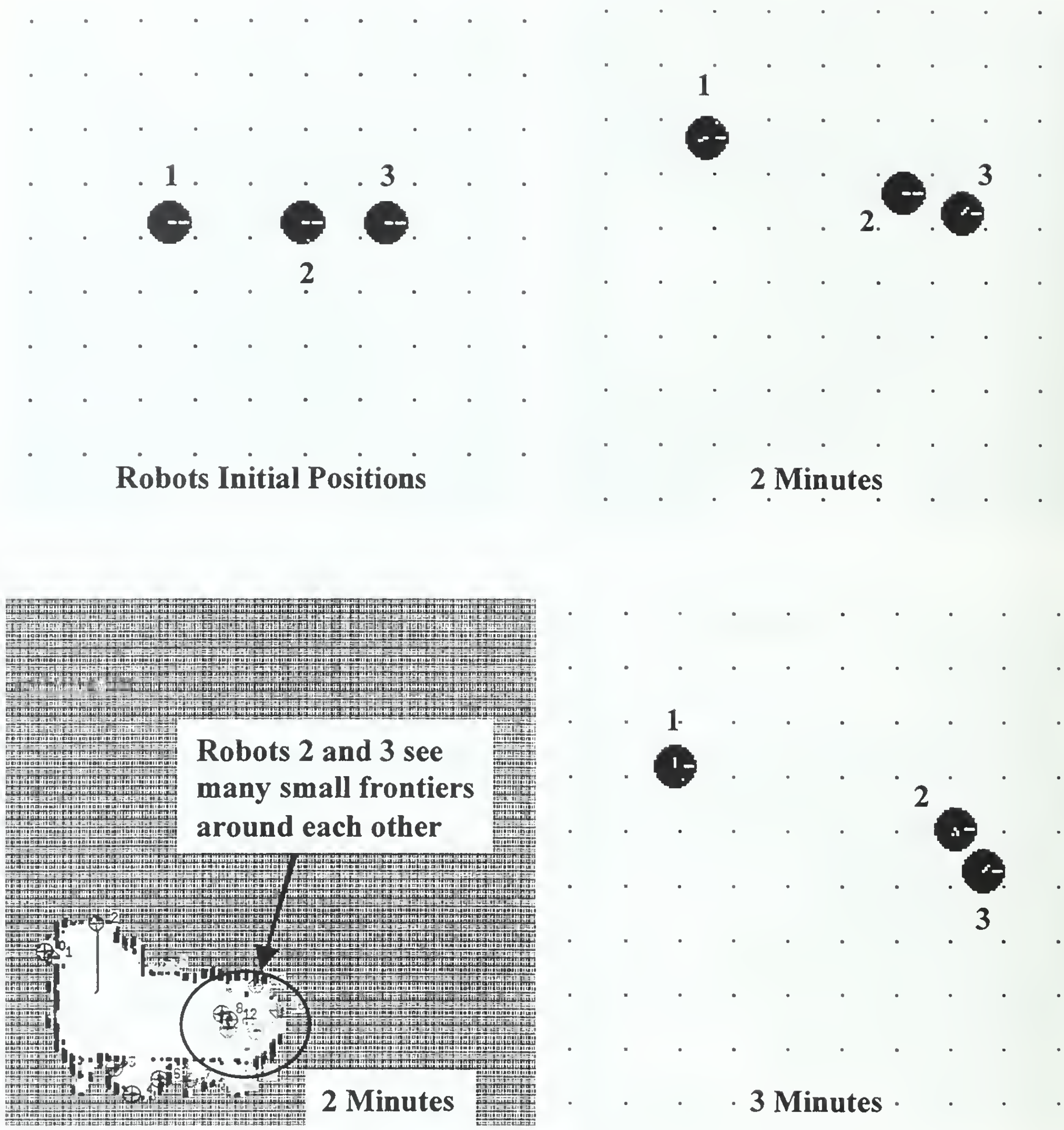


Figure 27. "Dancing Robot" behavior from robots in near proximity.

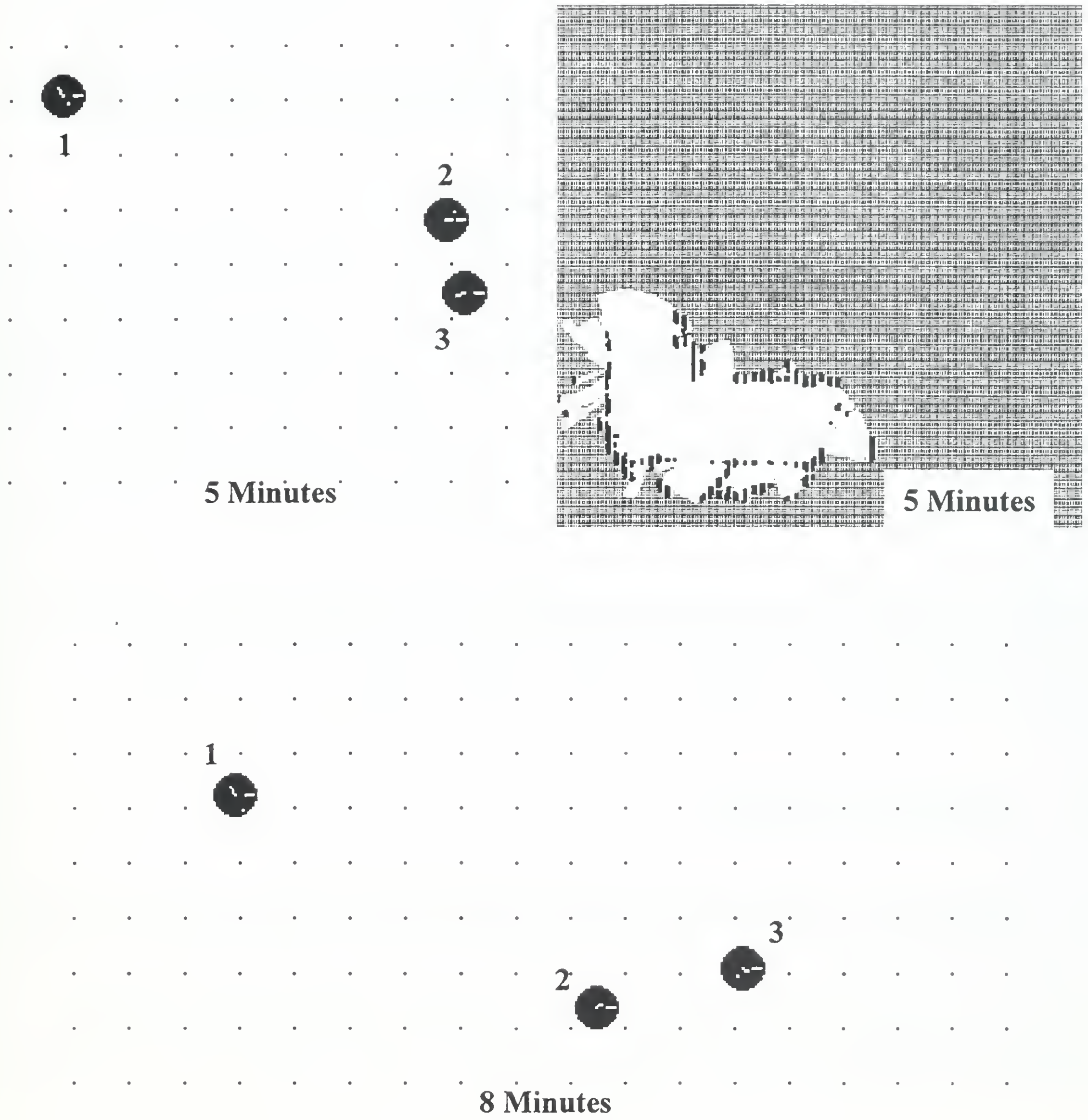


Figure 27 continued. "Dancing Robot" behavior from robots in near proximity.

c. *Propagation of "Bad" Data*

Throughout all the multiple robot trials it was evident that a single robot could enter "bad" data into the system. Localization reliability varied greatly from robot to robot and trial run to trial run depending on many variable such as the area being mapped, wheel slippage, etc. As seen in many of the multiple robot results shown here a

map made by multiple robots can be very accurate in some areas and very inaccurate in others depending on the varying quality of mapping data sent from the individual robots.

d. Network Reliability Problems

Throughout the trial runs there were unexplained network problems that seemed to increase in severity as the number of robots used was increased. Despite many attempts to track and mitigate the problem they continued to greatly detract from the ability to operate three or more robots for extended periods of time. For three robots 20 minutes was normally the longest time the robots would operate before packet errors caused termination of the experiment. This problem remains under investigation.

C. LESSONS LEARNED

There were several immediate lessons learned from this initial research. The first of these was that rotational odometry errors are much more detrimental to mapping efforts than translational errors. While translational errors do affect the quality of the map the robot is still able to navigate. Rotational odometry errors quickly increase to the point that that the robot is completely confused as to which direction it is facing and further navigation becomes impossible. However, any rotational localization scheme (such as wall or corner detection) will probably have a beneficial side effect of aiding translational localization as well.

The second lesson is that the better a robot can explore and map on an individual basis the better it will function as part of a multiple robot exploration and mapping

system. This is basically common sense. Continued improvements in single robot mapping will also improve multiple robot mapping.

The third lesson is that the network reliability issue needs further investigation. It needs to be determined whether the robot trials were causing the problem or if the cause was from an outside source. As mentioned above the local network administrators are currently investigating this problem.

VII. RECOMMENDATIONS FOR FURTHER STUDY

Due to the extensive amount of software modifications necessary and the constrained equipment availability there were a number of areas of research which promised to be very interesting, but which there was not time to pursue. It is hoped that future students will take up the task of continuing some of the possible avenues mentioned here now that the initial work has been done in order to provide a testbed system for research. These future research possibilities can be broken up into two main categories: those that would involve mainly software modifications only and those that would involve hardware additions or modifications in addition to software changes.

A. SOFTWARE CHANGES ONLY

Many possible research areas would require only software changes to the existing code and require no additional hardware. Also, there is still ample opportunity for optimization of the existing frontier-based exploration routines as currently implemented.

1. Centralized Map Building Process

There are many possible methods to centralize the map building process and possibly reduce or eliminate some of the counterproductive behavior seen in the initial trials while still allowing the individual robot processes to function with relative autonomy. One possibility is to implement a sort of "Blackboard" to which the individual robot processes would write, or send, map information.

The Blackboard would be a separate process or perhaps a "virtual" robot that would only accept local remote maps from all the robots and control no individual robot of its own. It would use the local maps sent to it to build a global map, which could then be sent back to the individual robot processes. It would take the place of the of the first robot process in the current implementation, acting as a server with all the individual robot processes as clients to it. How this might look for a system with four individual robots is illustrated in Figure 28.

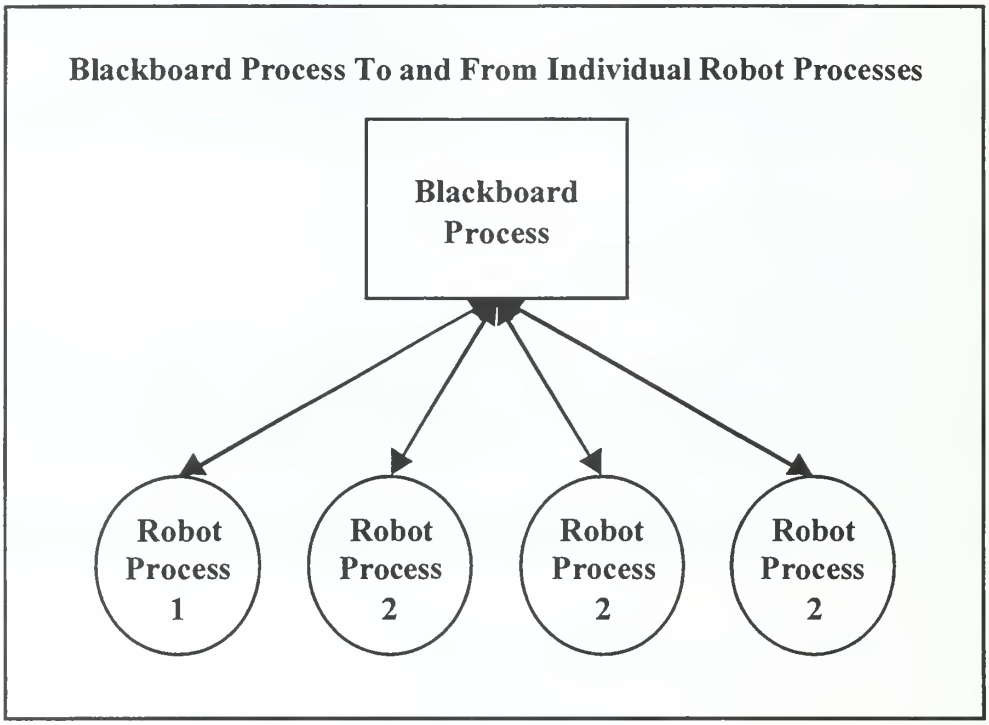


Figure 28. Illustration of a Blackboard-type process interacting with four individual robot processes.

This procedure could have a beneficial effect on the map building process in a number of different ways. By using only local scans to build the global map many of the problems of temporary obstacles gaining persistence and the unwanted reinforcement of "noisy" data are mitigated if not eliminated. Also, since the Blackboard process would

not be controlling a robot directly it can scan the shared memory location or "listen" for new local map messages constantly. This eliminates entirely the problem of missing local updates and losing map information from the individual robots. The Blackboard process would also provide a central point from which a user or operator could monitor the robotic mapping efforts of both the system and individual robots.

The Blackboard process could also be used to track the location of individual robots. Using this information the global map could be marked to either show the area physically covered by a robot as obstacle free or already explored. When this global map is sent back to the individual robots this information could be used to eliminate the problem of robots sensing each other as obstacles and the "dancing robot" behavior that follows. Having the Blackboard process create the global map also makes for a convenient location for any new robots joining the system to find out the most current map and avoid duplication of earlier efforts. More of the challenges of managing a dynamic robot population are discussed below.

The basics of the Blackboard process should actually require very little coding. Most of the functionality of taking in local maps and creating a global map is already done in the current implementation by the first robot process when acting as a server. Also, the client robot processes already write their local maps to and read the global map from the same shared memory location. For a basic Blackboard process simply removing the code that controls an individual robot from the original robot code and having the process constantly scan for and process local maps from the client processes would be sufficient.

2. Centrally Coordinated Effort

Another aspect that is worth investigating is removing some autonomy from the individual robot processes and creating some sort of centralized control or supervisory function. In this case the supervisory process would be able to direct or at least influence the individual robots' actions. This control could be constant, thus removing all autonomy from the individual robot processes, or on an as needed or exceptional basis, otherwise allowing the robots to act independently.

This control process would most likely act in conjunction with some sort of robot tracking process, perhaps one much like the Blackboard process described above. Besides eliminating the "Dancing Robots" and "Follow The Leader" behaviors it would also provide a single point from which an outside user or operator could direct the actions of an individual or groups of robots as well as see the results of the robotic mapping efforts. This is an important option in a deployable reconnaissance system.

This modification of the original implementation would require more extensive changes than just adding a simple tracking system. Besides the creating the supervisory process itself, it would also be necessary to make extensive modifications to the individual robot processes to have them accept commands from an outside source.

3. Dynamic Robot Population

The current implementation does not allow for an easy or simple method of joining additional robots to the system after initialization or for a way for a robot to

gracefully leave the system (say to go on to another task or report for maintenance). What is needed is some way to easily manage a dynamic or changing robot population.

Whenever a robot enters the system it needs some sort of unique identifier within the system so that other robots and any supervisory or other processes that exist have a way to identify or track the new robot. This identifier also serves to identify any local maps made by the robot. Under the current implementation the robot identifier is assigned by the *Nserver* in the order that the robots are created in the program. The user then assigns the individual robot processes a number corresponding to the one given by the *Nserver* program. For a dynamic robot population a dynamic method of allocating unique identifiers to robots is required.

One possibility would be a to use a simple Boolean array stored in the shared memory location used by all the robots. The size of the array would be the maximum number of robots that the system could manage. The array would be initialized to all zeros representing no robots in the system. As robots enter the system they would first scan the array until they found the first zero position. The robot would set the zero to a one and the numeric position in the array of that zero would become the robot's unique identifier.

Likewise, a robot leaving the system would reset its identifier position in the array to a zero, thus opening up that identifier for a new robot joining the system to use. If the system is filled with all the robots it can use or manage new robots would find the array filled with ones and would act appropriately depending on the system design, either waiting until an opening is available, moving on, or taking some other action altogether.

How this might function for a system with a maximum capacity of four robots is illustrated in Figure 29.

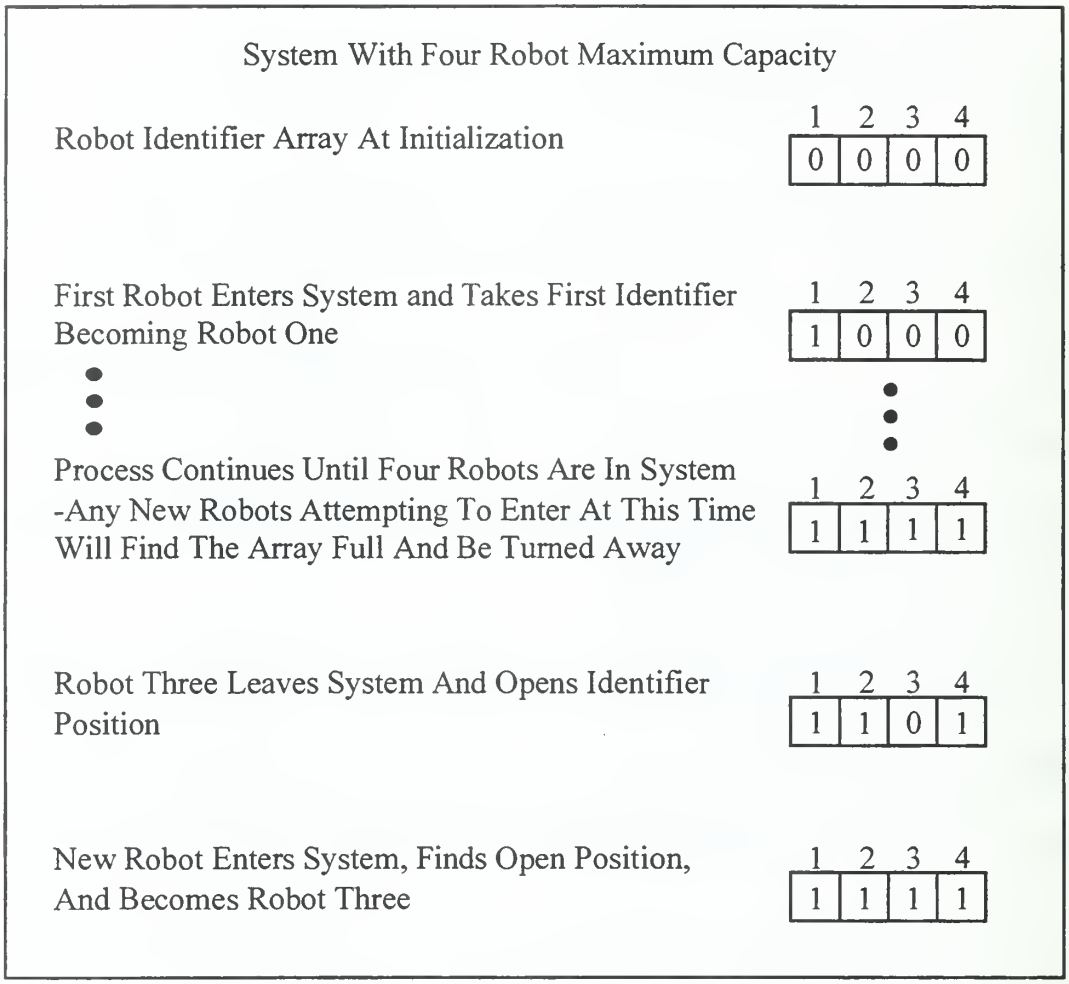


Figure 29. Illustration of the use of a Robot Identifier Array in the managing of a system with a four robot limit.

Furthermore, this array could hold other information rather than just the individual robot identifier data. In conjunction with the sort of supervisory process described above it might also be useful to have additional information such as sensor types and ranges available on the robot, mobility platform capabilities and limitations, and other types of data that could aid the central controlling process in managing the robotic resources available to it.

4. Communications Networking Model

The communications model in the current implementation is based on a point-to-point system in which an individual robot process explicitly communicates with only one other robot process at a time across a hardwired network connection. To better simulate a deployable system, where all links are wireless, a broadcast communications model should be used. One possibility would be to simulate the effects of range loss by including the individual robot's coordinates in the global map as an attachment to any message. The receiving robot could compare the sending robot's location to its own and decide whether or not to "accept" the message based on the distance between the two robots.

Another area worth further research is the general appropriateness of TCP as a communications protocol for mobile robot systems. Mobile robots in a real world wireless environment do not fit well with the design behind of TCP and its orientation around continuous streams of data. Mobile robots require a more message-based communications design. There are a number of other possible networking models and protocols other than the TCP/IP model currently used. Some work in this area has already been done, focusing in on the use of the User Datagram Protocol. [Ref. 33]

5. Improved Localization Method

As has already been mentioned the current implementation has no method of localization beyond simple dead reckoning using the robot's on-board odometric systems.

As shown in Chapter VI, this quickly proves insufficient as a means to accurately track the robot's location and map making efforts suffer accordingly. There has already been much work done with localization routines on a NOMAD 200 robot at NRL, both alone and in conjunction with frontier-based exploration [Ref. 20, 22]. It is hoped that their methods might be adapted to work with the NOMAD SCOUT robot as well.

Other research at NPS has concentrated on determining a robot's location in the real world through interpretation of its surroundings [Ref. 8, 9]. This work also shows promise of forming the basis of a general localization routine for any number or type of mobile robots. Other efforts have attempted to have the robot match its current surroundings to an evidence grid built by the robot and correct any rotational or translational errors that may have developed [Ref. 34].

Another possibility involves the outside use of some sort of supervisory process such as described above. This process could monitor a robot's self-reported location, the robot's reported surrounding, as well as any sensor reports from other robot's in the area. Using this information the supervisory process might track each of the robots in the system and send correction information to them as their dead reckoning systems begin to drift. Still another possibility is a group of robots forming a local reference system without the use of any central process. Some work has already been done in this area [Ref. 35].

6. Managing a Heterogeneous Robot Population

The multiple robot frontier-based exploration system has been implemented separately on groups of NOMAD 200 and NOMAD SCOUT robots. So far there has been no work done on integrating a heterogeneous population of robots in such a system. There are important questions concerning the code base for such a system. Would it be better (or even feasible) to write a single set of code that would adequately work with the varying sensor systems and mobility features of each platform? Or would it be better to have two completely different sets of routines for each type of robot?

An interesting aspect of a heterogeneous population of robots is the varying capabilities of each type of robot. While the NOMAD 200 has a more precise positioning capability, the NOMAD SCOUT is somewhat smaller and may be able to explore spaces the NOMAD 200 cannot reach. It would be interesting to find a way to use the diversity of the robot types as an advantage in accurate and complete exploration and map making. Both NRL and NPS now have both types of robots, thus providing a basis for such work. Some theoretical work has already been done in this area [Ref. 36].

7. Identifying System Tradeoffs

Initial research has already identified some of the tradeoffs of optimizing or adjusting various aspects of the system. For instance, reducing the "trustworthy" sonar range results in much less problems with specular reflections, but increases the total distance a robot or number of robots must travel in order to map a given area. This

increased travel distance, especially the larger number of small movements necessary to sweep the sonar sensors at new frontiers, leads to increased odometry error. On the other hand, increasing the "trusted" sonar range reduces travel requirements, but causes a corresponding increase in false sonar returns.

Much more study is needed on optimizing the sensor model for the best combination of accurate mapping with minimal movement. Of course an adequate localization routine would solve many problems, but even with a good localization routine minimizing travel distances would be beneficial to the overall system. Other possible tradeoff studies include autonomy versus centralized control and heterogeneous versus homogeneous robot populations. Some work has already been done in the study of the interaction between quantity of robots in a system, sensor quality, and mobility constraints on system performance for a given mission [Ref. 37].

8. Modified Movement Behaviors

The original movement behaviors for the robot in the routine *robot.cc* were written with the capabilities of the NOMAD 200 robot in mind. Problems arise with using these same behaviors on the NOMAD SCOUT robot. While the NOMAD 200 can translate on its own axis, the NOMAD SCOUT cannot. Therefore movement behaviors that would have been simple rotations or shallow arcs on a NOMAD 200 become much larger movements when the conversion macros interpret them for use on the NOMAD SCOUT.

This causes many difficulties when the NOMAD SCOUT is near to and facing a wall or other obstacle at the end of a sensor sweep. When the robot tries to move on to a

new frontier the macro translates the current movement commands into a forward turning motion. If the robot is too close to the wall it will be blocked and eventually the new frontier it should have traveled to will be marked as inaccessible. Some type of backing up or modified turn behavior would seem to be a possible solution.

Other opportunities also exist to counter the “Follow The Leader” and “Dancing Robot” behaviors. One possible solution to be explored might be to have the robot broadcast its location either to all robots in the system or at least those nearby. That way a robot could recognize that the obstacle it is trying to map is in fact another robot. Of course any sort of centralized supervisory process could also counter this problem very easily. Another possibility is to set some sort of limit on how long a robot would continue to map a small area despite new frontiers constantly appearing in that area. After a period of time it could give up and move on to a radically different geographical area.

B. HARDWARE AND SOFTWARE CHANGES

Other possible research areas would require additional hardware and/or modification of the already existing hardware in the system. In addition, software modifications would be necessary in order to use the added or changed hardware. Some of the hardware for the research possibilities mentioned below is already available at NPS.

1. Human – Robotic System Interaction

The area of real-time human-robotic interaction holds many, many possible research opportunities. In the current implementation the current map is displayed on a workstation and the opportunities for user interaction are very limited. Obviously, for a practical, deployable system these limitations must be removed.

There is a need for some sort of portable system through which a user can receive information from a single robot or groups of robots and also direct the actions of an individual robot or number of robots. Ideally, the device would be lightweight, unobtrusive, and user friendly in design and use. While earlier research had focused on rather large control and display systems [Ref. 38], perhaps the best model available today for such a device is the in the form factor and design of a Personal Digital Assistant (PDA).

There are a number of different types of PDAs available for research at NPS. Many of them have some type of wireless connection or network capability. What is required is a method for a map data to be transmitted to these devices in a useable format and some method for operator commands to be sent back to the network and then on to the robot(s). Once this is possible many other research opportunities become available. What is the best way to manage the system from the operator point-of-view? For a deployed system, how much control does the operator need or even want? Is controlling the robot(s) the operator's primary duty or something that should be done on an as-required basis? What are the possibilities for cooperative exploration of an area between

human and robot? How best should the system signal the operator to possible danger areas or other areas of interest? These and other questions should be investigated early before large amounts of funding are spent on programs that later are found to be unworkable or impractical to implement.

2. Outdoor Trials

Any practical, field deployable system will need to work outdoors as well indoors. Even if the system is build primarily for mapping the interiors of buildings the individual robots will most likely have to traverse rough, broken, urban terrain to travel from one operating site to another. Neither the NOMAD 200 nor the NOMAD SCOUT has much capability in this regard as currently configured. At this time their manufacturer has no announced plans to market any outdoor-capable models either.

However, there does exist other outdoor-capable robotic systems at NPS. In particular the “Shepherd” vehicle [Ref. 39, 40], under cooperative development by several different departments at NPS, is a robot with a four-wheeled all-terrain-vehicle (ATV) style chassis with independent driving and steering capability. Much study has been conducted on this platform concerning motion control and localization via inertial sensors and the use of a Global Positioning System (GPS) receiver. It would seem that integrating some sort of mapping capability into it would be a natural next step. There are many questions about how much of the current implementation of the frontier-based exploration code could be ported to the new platform, but the fundamentals of the process would seem to remain the same.

3. Removing Dependency on Wired Network

Perhaps one of the most ambitious possibilities is removing the dependency that the current implementation has on a wired network to provide communications connectivity. The NOMAD SCOUT has the capability to be completely independent through the use of a laptop computer running the LINUX operating system. A laptop can be mounted on top of the NOMAD SCOUT and can run the same code locally (after it is recompiled) that is currently run on a remote Sun workstation.

Once the NOMAD SCOUT robots are operating independently of a wired network it might be possible to better implement a broadcast model of communication amongst the robots. The wireless modems used in the current implementation are capable of serving as either point-to-point communications stations or as part of a distributed system. Because there would no longer be a shared memory location, the exploration and mapping software would have to be modified to actually send all the map data and not just a pointer to the data or message that it is available to the other robots in the system. There is already a body of work supporting communications protocols for distributed robotic systems without a centralized communications server [Ref. 41].

4. Additional Sensor Systems

Using laptops on the NOMAD SCOUT robots as mentioned above also opens up the opportunity to integrate additional sensor systems on the platform. The unused input ports of the laptop provide a means to include video, audio, or any of a number of

other sensing devices to the system. In addition, there is also the possibility of adding a means of detecting beacons in the environment or on other robots as an aid to navigation and localization.

VIII. CONCLUSIONS

Oftentimes the hardest part of any journey is just getting started. This thesis and the research involved in creating it have been aimed at creating a starting point for future studies. Now that the basics of a real world (as compared to simulated) multiple robot system has been developed and implemented at NPS, a huge number of additional avenues of investigation are available.

It has been shown that much work remains in order to create a consistent and robust exploration and mapping system before many of the questions surrounding robotic battlefield support can be answered. However, now there exists at NPS a “critical mass” of mobile robot types with varying capabilities and at least basic software to enable some of them to operate in a shared real-world environment toward a common goal.

Many of the problems encountered in this research are very similar to those discovered by other researchers when moving a robotic system from one environment to another [Ref. 42]. In the case of this research the testing of multiple mobile robots has been moved from simulation-only environment to a combination of simulation and real-world testing. This transition has revealed many details of multiple robotic systems that otherwise would have remained hidden in simulator-only testing.

In Chapter VI there are listed many possible areas of study based on the work presented here. These are just the start of many possible thesis opportunities involving hardware, software, human-machine interaction, etc. One of the most exciting and challenging things about robotics as a field of study is the number of different fields and

disciplines that it encompasses. It is hoped that future students will take up this challenge and carry on the work started here.

APPENDIX A. SOURCE CODE FOR COLLECTION OF SIMPLE SENSOR RETURN DATA

This appendix contains the source code that was used to collect sonar range data on early map making efforts with the NOMAD SCOUT robot.

```
/************************************************************
 *
 * PROGRAM: world_sonar.c
 *
 * PURPOSE: To collect sonar data for establishing a world map.
 * modified for Scout by Patrick A. Hillmeyer
 ************************************************************/


/*** Include Files ***/

#include "Nclient.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>



/***  Conversion MACROS courtesy of Nomadic Inc   ***/
/** original beta macros for SCOUT models  ****/

#define RIGHT(trans, steer)  (trans + (int)((float)steer*368.61/3600.0))
#define LEFT(trans, steer)   (trans - (int)((float)steer*368.61/3600.0))

#define scout_vm(trans, steer)   vm(RIGHT(trans, steer), LEFT(trans,
steer), 0)
#define scout_pr(trans, steer)   pr(RIGHT(trans, steer), LEFT(trans,
steer), 0)




/*** Function Prototypes ***/

void GetSensorData(void);


/*** Globals ***/

long SonarRange[16];  /* array of sonar readings (inches) */
long IRRange[16];  /* array of infrared readings (no units) */
long robot_config[4];

/*** Main Program ***/

main (unsigned int argc, char** argv)
{
  int i, j, index;
  int order[16];
  FILE *fp;

```

```
  /* Connect to Nserver. The parameter passed must always be 1. */
  connect_robot(1, MODEL_SCOUT, "scout1.ece.nps.navy.mil", 4001);


  /* Initialize Smask and send to robot. Smask is a large array that
  controls which data the robot returns back to the server. This
  function tells the robot to give us everything. */
  init_mask();


  /* Configure timeout (given in seconds). This is how long the robot
  will keep moving if you become disconnected. Set this low if there
  are walls nearby. */
  conf_tm(1);


  /* Sonar setup  */
  for (i = 0; i < 16; i++)
    order[ i] = i;
  conf_sn(15,order);


  zr();  /* tell robot to zero itself */



  fp = fopen("range.dat", "w");

  /* Main loop. */
  for (i=0; i<2; i++)
    {
      GetSensorData();

      for (j=0; j<16; j++)
      fprintf(fp, "%8d %8d %8d %8d %8d  \n",
             robot_config[ 0],robot_config[ 1],robot_config[ 2],
             robot_config[ 3],
             SonarRange[ j] );
    }

  fclose(fp);

  /* Disconnect. */
  disconnect_robot(1);
}



/* GetSensorData(). Read in sensor data and load into arrays. */
void GetSensorData (void)
{
  int i;


  /* Read all sensors and load data into State array. */
  gs();


```

```
    /* Read State array data and put readings into individual arrays. */
    for (i = 0; i < 16; i++)
      {
        /* Sonar ranges are given in inches, and can be between 6 and
        255, inclusive. */
        SonarRange[i] = State[17+i];

        /* IR readings are between 0 and 15, inclusive. This value is
        inversely proportional to the light reflected by the detected
        object, and is thus proportional to the distance of the
        object. Due to the many environmental variables effecting the
        reflectance of infrared light, distances cannot be accurately
        ascribed to the IR readings. */
        IRRange[i] = State[1+i];
      }


    for (i = 0; i < 4; i++)
      robot_config[i] = State[34+i];
  }
```

APPENDIX B. MATLAB SOURCE CODE FOR PLOTTING OF SIMPLE SENSOR RETURN DATA

This appendix contains the MATLAB M-file that was used to analyze and display the sonar range data from early map making efforts with the NOMAD SCOUT robot.

```
% Capt Patrick A. Hillmeyer, USMC
% Code originally written for EC 4300 Robotics class
% Written to interpret sonar range data collected
% from a NOMAD SCOUT robot

% Load the range data collected during the robot's travel
load range1.dat

robo_data=range1;

% Convert robot x,y coordinates to inches
rob_x_in_world=robo_data(:,1)/10;
rob_y_in_world=robo_data(:,2)/10;

% In this data the base and turret are aligned
% therefore no alignment correction is necessary
% Carryover from old NOMAD 200 version of code

% covert angles to degrees then to radians
base_angle=(robo_data(:,3)/10)*pi/180;
obj_dist_fr_rob=robo_data(:,5);

num_sensors=16;
deg_per_sensor=360/num_sensors;
rad_per_sensor=deg_per_sensor*pi/180;

% correct for sensor location offset
% from robot center
rob_radius=8.81;

% Set the range at which to trust the
% sonar data
s_trust=60;

% plot robot path alone
figure(1)
plot(rob_x_in_world,rob_y_in_world,'w.')
title('Robot path in real (or simulated) world')
xlabel('Inches'),ylabel('Inches')
axis('equal')

% now plot sonar hits as robot moved
x_sonar_hits=[];
y_sonar_hits=[];

% Read through the data in sets corresponding
% to the number of sensor readings taken at each
% location in the robot's path
for ctr2=0:((length(robo_data)/num_sensors)-1)

 for ctr3=1:num_sensors
```

```
   abs_data_pt=(num_sensors*ctr2)+ctr3;

   % only process if valid reading
   if obj_dist_fr_rob(abs_data_pt)<s_trust

    A_B_T=[ cos(base_angle(abs_data_pt)) ...
           -sin(base_angle(abs_data_pt)) ...
           0 rob_x_in_world(abs_data_pt);
           sin(base_angle(abs_data_pt)) ...
           cos(base_angle(abs_data_pt)) ...
           0 rob_y_in_world(abs_data_pt);
           0 0 1 0;
           0 0 0 1];

    % correct for off-by-one discrepency
    % in sensor numbering
    sensor_num=ctr3-1;

    B_P=[  (obj_dist_fr_rob(abs_data_pt)+rob_radius) ...
          *cos(sensor_num*rad_per_sensor);
          (obj_dist_fr_rob(abs_data_pt)+rob_radius) ...
          *sin(sensor_num*rad_per_sensor);
          0;
          1];

    A_P=A_B_T*B_P;

    x_sonar_hits=[ x_sonar_hits A_P(1)];
    y_sonar_hits=[ y_sonar_hits A_P(2)];

   end  % end for if

  end % end for ctr3

 end % end for ctr2

 figure(2)
 plot(x_sonar_hits,y_sonar_hits,'w.', ...
      rob_x_in_world,rob_y_in_world,'w.')
 title('Simulated world sonar data - 60 inch sonar reliability')
 xlabel('Inches'),ylabel('Inches')
 axis('equal')
```

APPENDIX C. FRONTIER-BASED EXPLORATION CODE – GRID.H

This appendix contains the header file for the routine that builds the evidence grid based on the sensor return data.

```
/*

  grid.h

  Header file for robot/evidence grid functions
  original code by Brian Yamauchi

  Modifications for SCOUT THESIS
  by Patrick A. Hillmeyer

*/

#include "cmacs.h"
#include "volsense.h"

/* Grid occupied threshold */

#define GRID_POS_THRESH 16

/* Grid unoccupied threshold */

#define GRID_NEG_THRESH -16

/* Local Grid dimensions (feet) */

/* BEGIN SCOUT THESIS CHANGE */
/* change the local grid dimensions to match the global grid dimensions
*/
#define X_MIN -22.0
#define X_MAX 22.0
#define Y_MIN -22.0
#define Y_MAX 22.0
#define Z_MIN 0.0
#define Z_MAX 5.0

/* Grid resolution (cells) */
/* increase the number of cells  */
/* the value here has to be a power of 2 and symetrical  */
/*  i.e 64 by 64, 128 by 128, etc    */
/* this is true for all the other grid resolutions below as well */

#define X_RES 256
#define Y_RES 256
#define Z_RES 1

/* END SCOUT THESIS CHANGE  */


/* Global grid dimensions (feet) */

#define GLOBAL_X_MIN -22.0
#define GLOBAL_X_MAX 22.0
```

```
#define GLOBAL_Y_MIN -22.0
#define GLOBAL_Y_MAX 22.0
#define GLOBAL_Z_MIN 0.0
#define GLOBAL_Z_MAX 5.0



/* Global grid resolution (cells) */

#define GLOBAL_X_RES 256
#define GLOBAL_Y_RES 256
#define GLOBAL_Z_RES 1



/* Navigation grid dimensions (feet) */

#define NAV_X_MIN -22.0
#define NAV_X_MAX 22.0
#define NAV_Y_MIN -22.0
#define NAV_Y_MAX 22.0
#define NAV_Z_MIN 0.0
#define NAV_Z_MAX 5.0



/* Resolution of navigation grid (cells) */

#define NAV_X_RES 256
#define NAV_Y_RES 256
#define NAV_Z_RES 1



/* Sensor modes */

#define SONAR_MODE 0
#define LASER_MODE 1
#define INTEG_MODE 2

/* Sensor parameters */

/* BEGIN SCOUT THESIS CHANGE  */

/* Scout and Scout2 dimensions 15.125 in sensor to sensor diameter  */
/* Scout2 sonar height 10.25 in  Scout close enough to use same value */
/* Height from floor to sonar (ft)  Scouts 10.25 in */
#define SONAR_HEIGHT 0.8542
/* Offset from robot center to sonar (ft)  Scouts 7.5625 in */
#define SONAR_RAD 0.63
/* Separation between adjacent sonars (deg)  - same as Nomad 200 */
#define SONAR_SEP 22.5
/* Height from floor to IR (ft)  - None on Scout */
#define IR_HEIGHT 0.0
/* Offset from robot center to IR (ft) - None on Scout */
#define IR_RAD 0.0
/* Separation between adjacent IR (deg) - None on Scout */
#define IR_SEP 0.0
```

```
/* Height from floor to laser (ft) - None on Scout */
#define LASER_HEIGHT 0.0
/*  END SCOUT THESIS CHANGE  */

 #define HEIGHT_OFFSET 0.0    /* z-axis offset (ft) */

/* Maximum sonar reading (indicates no reflection) */
#define MAX_SONAR_READING 255

/* Maximum (valid) sonar range (feet) -- Use 21.25 for no truncation */

/* BEGIN SCOUT THESIS CHANGE  */

/* This is the trustworthy range of the sonar in ft */
/* shorter range for Scout to reduce specular reflection problem */
#define MAX_SONAR_RANGE 8.0

/*#define MAX_SONAR_RANGE 10.0*/
/*#define MAX_SONAR_RANGE 21.25*/

/* Maximum sonar range for occupied cells (feet) */

/* This value seems to have no effect */
/*#define MAX_SONAR_OCC_RANGE 3.0*/
#define MAX_SONAR_OCC_RANGE 15.0

/* Maximum IR reading (indicates no reflection) */

#define MAX_IR_READING 0     /* No IR on Scout */

/* Maximum (valid) laser range (feet) */

/*#define MAX_LASER_RANGE 100.0*/
#define MAX_LASER_RANGE 0.0    /* No laser on Scout */

/* END SCOUT THESIS CHANGE */

/* Size of cell in robot window */

#define DISPLAY_SCALE 56.25

/* Angle conversion constants */

#define M_RAD2DEG 57.29578
#define M_DEG2RAD 0.017453293

/* Laser configuration parameters */

#define LASER_MODE_OFF 0x32      /* 1 100 1 1 */   /* X,Y pairs */
#define LASER_MODE_ON 0x33      /* 1 100 1 1 */   /* X,Y pairs */
#define LINE 0x03          /* 0 000 1 1 */   /* X,Y pairs for endpoints
*/
#define THRESHHOLD 70      /* f4=30, f2.8=5,  factory=20 */
#define WIDTH 40           /* f4=20, f2.8=20, factory=20 */
#define NUMDATA 120        /* Number of points returned */
#define AVG 1              /* Number of pixels averaged */

/* Stepsize for printing grid */
```

```

#define PRINT_STEP 1


/* Robot size */

/* BEGIN SCOUT THESIS CHANGE */
/* Scout2 is taller use its value - 14 in */
/* Add bumper space for total radius */

/* Robot radius (feet)  Scout sensor radius plus .756 in for bumpers*/
#define ROBOT_RADIUS 0.693

/* Robot height (feet)  use Scout2 14 in */
#define ROBOT_HEIGHT 1.1667

/* Size necessary for safe robot passage (feet) */
#define ROBOT_PASSAGE_RADIUS 0.7    /* Add small safety margin */

/* END SCOUT THESIS CHANGE */


/* Grid decay factor */

#define GRID_DECAY 8

/* Grid translation parameters */

#define NUM_TRANS 1           /* Number of translations in each
direction
                                   along each axis */
#define TRANS_STEP 0.2        /* Size of each translation step (feet) */

/* Grid rotation parameters */

#define NUM_ROT 1       /* Number of rotations (in each direction) */
#define ROT_STEP 2.0          /* Rotation step (degrees) */

/* Mimimum change in position (1/10 inch) to update */

#define MIN_DELTA 46.88

/* Relative weight of clear cells in fine grid to coarse grid conversion
*/

#define F2C_CLEAR_WT 1

/* Relative weight of occupied cells in fine grid to coarse grid
conversion */

#define F2C_OCC_WT 4

/* Maximum laser/sonar angle difference for laser-limited sonar
(degrees) */

#define LLS_MAX_ANGLE_DIFF 3.0
```

APPENDIX D. FRONTIER-BASED EXPLORATION CODE – GRID.C

This appendix contains the source code for the routine that builds the evidence grid based on the sensor return data.

```
/*

  grid.c

  Robot/evidence grid functions
  original code by Brian Yamauchi

  Modification for SCOUT THESIS
  by Patrick A. Hillmeyer

*/

#include <stdio.h>
#include <math.h>
#include "Nclient.h"
#include "grid.h"

double min3(double x, double y, double z)
/*
  Return the minimum of three values
 */
{
    double m;     /* Minimum */

    m = x;
    if (y < m) {
      m = y;
    }
    if (z < m) {
      m = z;
    }
    return(m);
}

int world2grid(Map3D map, double wx, double wy, double wz,
               int *gx, int *gy, int *gz)
/*
  Return grid coordinates for location in world coordinates
 */
{
    double xsize, ysize, zsize;           /* Size of grid cell */

    if ((wx < map.lomv[0]) || (wx > map.himv[0]) ||
      (wy < map.lomv[1]) || (wy > map.himv[1]) ||
      (wz < map.lomv[2]) || (wz > map.himv[2])) {
/*    printf("world2grid: point (%f, %f, %f) out of range <%f:%f, %f:%f,
%f:%f>.\n",
               wx, wy, wz, map.lomv[0], map.himv[0], map.lomv[1],
map.himv[1],
               map.lomv[2], map.himv[2]);*/
      return(-1);
    }
```

```
    xsize = (map.himv[ 0]  - map.lomv[ 0] ) / map.msize[ 0] ;
    ysize = (map.himv[ 1]  - map.lomv[ 1] ) / map.msize[ 1] ;
    zsize = (map.himv[ 2]  - map.lomv[ 2] ) / map.msize[ 2] ;

    *gx = (int) ((wx - map.lomv[ 0] ) / xsize);
    *gy = (int) ((wy - map.lomv[ 1] ) / ysize);
    *gz = (int) ((wz - map.lomv[ 2]  + HEIGHT_OFFSET) / zsize);

    if ((*gx < 0) || (*gx >= map.msize[ 0] ) ||
      (*gy < 0) || (*gy >= map.msize[ 1] ) ||
      (*gz < 0) || (*gz >= map.msize[ 2] )) {
/*    printf("world2grid: world location (%f, %f, %f) --> cell [ %d, %d,
%d] out of range.\n", wx, wy, wz, *gx, *gy, *gz);*/
      return(-1);
    }

    return(1);
}

int world2index(Map3D map, double wx, double wy, double wz)
/*
  Return grid cell index for location in world coordinates
 */
{
    double xsize, ysize, zsize;           /* Size of grid cell */
    int gx, gy, gz;                    /* Coordinates of grid cell */
    int index;                         /* Grid cell array index */

    if ((wx < map.lomv[ 0] ) || (wx > map.himv[ 0] ) ||
      (wy < map.lomv[ 1] ) || (wy > map.himv[ 1] ) ||
      (wz < map.lomv[ 2] ) || (wz > map.himv[ 2] )) {
/*    printf("world2index (%f, %f, %f) out of range <%f:%f, %f:%f,
%f:%f>.\n",         wx, wy, wz, map.lomv[ 0] , map.himv[ 0] , map.lomv[ 1] ,
map.himv[ 1] ,
               map.lomv[ 2] , map.himv[ 2] );*/
      return(-1);
    }

    xsize = (map.himv[ 0]  - map.lomv[ 0] ) / map.msize[ 0] ;
    ysize = (map.himv[ 1]  - map.lomv[ 1] ) / map.msize[ 1] ;
    zsize = (map.himv[ 2]  - map.lomv[ 2] ) / map.msize[ 2] ;

    gx = (int) ((wx - map.lomv[ 0] ) / xsize);
    gy = (int) ((wy - map.lomv[ 1] ) / ysize);
    gz = (int) ((wz - map.lomv[ 2]  + HEIGHT_OFFSET) / zsize);

    index = gz * map.msize[ 0]  * map.msize[ 1]  + gy * map.msize[ 0]  + gx;

    if ((index < 0) || (index >= map.msize[ 0]  * map.msize[ 1]  *
map.msize[ 2] )) {
/*    printf("world2index: world location (%f, %f, %f) --> index [ %d]
out of range.\n", wx, wy, wz, index);*/
      return(-1);
    }

/*    printf("world2grid: world location (%f, %f, %f) --> cell [ %d, %d,
%d] <%d>.\n", wx, wy, wz, gx, gy, gz, index);*/
```

```
    fflush(stdout);

    return(index);
}

void grid2world(Map3D map, int gx, int gy, int gz,
            double *wx, double *wy, double *wz)
/*
  Return world coordinates for location in grid coordinates
 */
{
    double xsize, ysize, zsize;           /* Size of grid cell */
/*    int tx, ty, tz;*/

    xsize = (map.himv[0] - map.lomv[0]) / map.msize[0];
    ysize = (map.himv[1] - map.lomv[1]) / map.msize[1];
    zsize = (map.himv[2] - map.lomv[2]) / map.msize[2];

    *wx = (double) (gx + 0.5) * xsize + map.lomv[0];
    *wy = (double) (gy + 0.5) * ysize + map.lomv[1];
    *wz = (double) (gz + 0.5) * zsize + map.lomv[2];

/*    if (world2grid(map, *wx, *wy, *wz, &tx, &ty, &tz) == -1) {
      printf("<%d, %d, %d> --> (%f, %f, %f) --> <???, ???, ???>\n",
             gx, gy, gz, *wx, *wy, *wz);
    }
    else {
      printf("<%d, %d, %d> --> (%f, %f, %f) --> <%d, %d, %d>\n",
             gx, gy, gz, *wx, *wy, *wz, tx, ty, tz);
    }*/
}

int grid2index(Map3D map, int gx, int gy, int gz)
/*
  Return grid cell index for grid cell coordinates
 */
{
    int index;                            /* Grid cell array index */

    index = gz * map.msize[0] * map.msize[1] + gy * map.msize[0] + gx;
    return(index);
}

void set_location(Map3D map, double x, double y, double z, int value)
/*
   Set probability of grid cell corresponding to world location
*/
{
    int gindex;                   /* Grid array index */

    gindex = world2index(map, x, y, z);
    if (gindex > -1) {
      map.mapm[gindex] = value;
    }
}

void set_grid(Map3D map, int x, int y, int z, int value)
/*
```

```
    Set probability of specified grid cell
*/
{
    int gindex;               /* Grid array index */

    gindex = z * map.msize[0] * map.msize[1] + y * map.msize[0] + x;
    if ((gindex < 0) || (gindex >= map.msize[0] * map.msize[1] *
map.msize[2])) {
/*     printf("set_grid: cell [%d, %d, %d] out of range <%d, %d, %d>.\n",
              x, y, z, map.msize[0], map.msize[1], map.msize[2]);*/
      return;
    }
    map.mapm[gindex] = value;
}

void grid_init(Map3D *map1,   /* Grid pointer */
               double cx, /* Center x-coord (feet) */
               double cy) /* Center y-coord (feet) */
/*
  Initialize evidence grid
*/
{
  double lov[3], hiv[3];      /* Grid corners (feet) */
  int msize[3];               /* Grid size (cells) */

  map1->cx = cx;
  map1->cy = cy;

  msize[0] = X_RES;
  lov[0] = cx + X_MIN;
  hiv[0] = cx + X_MAX;

  msize[1] = Y_RES;
  lov[1] = cy + Y_MIN;
  hiv[1] = cy + Y_MAX;

  msize[2] = Z_RES;
  lov[2] = Z_MIN;
  hiv[2] = Z_MAX;

  MakeMap3D(msize, lov, hiv, map1);
}

void grid_init_global(Map3D *map1,  /* Grid pointer */
                      double cx,  /* Center x-coord (feet) */
                      double cy)  /* Center y-coord (feet) */
/*
  Initialize global evidence grid
*/
{
  double lov[3], hiv[3];      /* Grid corners (feet) */
  int msize[3];               /* Grid size (cells) */

  map1->cx = cx;
  map1->cy = cy;

  msize[0] = GLOBAL_X_RES;
  lov[0] = cx + GLOBAL_X_MIN;
```

```
  hiv[ 0] = cx + GLOBAL_X_MAX;

  msize[ 1] = GLOBAL_Y_RES;
  lov[ 1] = cy + GLOBAL_Y_MIN;
  hiv[ 1] = cy + GLOBAL_Y_MAX;

  msize[ 2] = GLOBAL_Z_RES;
  lov[ 2] = GLOBAL_Z_MIN;
  hiv[ 2] = GLOBAL_Z_MAX;

  MakeMap3D(msize, lov, hiv, map1);
}

void grid_init_nav(Map3D *map1,     /* Grid pointer */
               double cx,       /* Center x-coord (feet) */
               double cy)       /* Center y-coord (feet) */
/*
  Initialize evidence grid for navigation
*/
{
  double lov[ 3], hiv[ 3];        /* Grid corners (feet) */
  int msize[ 3];                  /* Grid size (cells) */

  map1->cx = cx;
  map1->cy = cy;

  msize[ 0] = NAV_X_RES;
  lov[ 0] = cx + NAV_X_MIN;
  hiv[ 0] = cx + NAV_X_MAX;

  msize[ 1] = NAV_Y_RES;
  lov[ 1] = cy + NAV_Y_MIN;
  hiv[ 1] = cy + NAV_Y_MAX;

  msize[ 2] = NAV_Z_RES;
  lov[ 2] = NAV_Z_MIN;
  hiv[ 2] = NAV_Z_MAX;

  MakeMap3D(msize, lov, hiv, map1);
}

void grid_print(Map3D map, int yaxis)
/*
  Print evidence grid occupancy probabilities
*/
{
  int x, y, z;                    /* Cell index */
  int xsize, ysize, zsize;        /* Grid dimensions (# cells) */
  int p;                    /* Occupancy probability */
  int empty;                      /* Empty level flag */

  xsize = map.msize[ 0];
  ysize = map.msize[ 1];
  zsize = map.msize[ 2];

  for (z = 0; z < zsize; z++) {
    y = x = 0;
    empty = 1;
```

```
    while((y < ysize) && (x < xsize) && empty) {
      if (map.mapm[ z * xsize * ysize + y * xsize + x] != 0) {
      empty = 0;
      }
      x++;
      if (x == xsize) {
      x = 0;
      y++;
      }
    }

    if (!empty) {
      printf("Level: %d\n\n", z);

      for (y = 0; y < ysize; y++) {
      for (x = 0; x < xsize; x++) {
        if (yaxis == 1) {
          p = map.mapm[ z * xsize * ysize + (ysize - y - 1) * xsize + x];
        }
        else {
          p = map.mapm[ z * xsize * ysize + y * xsize + x];
        }
        if (p > 0) {
          printf("#");
        }
        else if (p == 0) {
          printf("?");
        }
        else if (p > -25) {
          printf(":");
        }
        else if (p > -50) {
          printf(".");
        }
        else {
          printf(" ");
        }
      }
      printf("\n");
      }
      getchar();
    }
  }
}

void sonar_print(Map3D map, int yaxis)
/*
  Print evidence grid occupancy probabilities for sonar level
*/
{
  int x, y, z;                  /* Cell index */
  int xsize, ysize, zsize;      /* Grid dimensions (# cells) */
  int p;                /* Occupancy probability */
  int empty;                    /* Empty level flag */

  xsize = map.msize[ 0];
  ysize = map.msize[ 1];
  zsize = map.msize[ 2];
```

```
  z = (int) ((SONAR_HEIGHT + HEIGHT_OFFSET - map.lomv[2]) /
          (map.himv[2] - map.lomv[2]) * zsize);

  printf("");
  for (y = 0; y < ysize; y += PRINT_STEP) {
    for (x = 0; x < xsize; x += PRINT_STEP) {
      if (yaxis == 1) {
      p = map.mapm[z * xsize * ysize + (ysize - y - 1) * xsize + x];
      }
      else {
      p = map.mapm[z * xsize * ysize + y * xsize + x];
      }
      if (p > 0) {
      printf("#");
      }
      else if (p == 0) {
      printf("?");
      }
      else if (p > -25) {
      printf(":");
      }
      else if (p > -50) {
      printf(".");
      }
      else {
      printf(" ");
      }
    }
    printf("\n");
  }
}

void laser_print(Map3D map, int yaxis)
/*
  Print evidence grid occupancy probabilities for sonar level
*/
{
  int x, y, z;                /* Cell index */
  int xsize, ysize, zsize;    /* Grid dimensions (# cells) */
  int p;                 /* Occupancy probability */
  int empty;                  /* Empty level flag */

  xsize = map.msize[0];
  ysize = map.msize[1];
  zsize = map.msize[2];

  z = (int) ((LASER_HEIGHT + HEIGHT_OFFSET - map.lomv[2]) /
          (map.himv[2] - map.lomv[2]) * zsize);

  printf("");
  for (y = 0; y < ysize; y += PRINT_STEP) {
    for (x = 0; x < xsize; x += PRINT_STEP) {
      if (yaxis == 1) {
      p = map.mapm[z * xsize * ysize + (ysize - y - 1) * xsize + x];
      }
      else {
      p = map.mapm[z * xsize * ysize + y * xsize + x];
```

```
      }
      if (p > 0) {
      printf("#");
      }
      else if (p == 0) {
      printf("?");
      }
      else if (p > -25) {
      printf(":");
      }
      else if (p > -50) {
      printf(".");
      }
      else {
      printf(" ");
      }
    }
    printf("\n");
  }
}

void grid_display(Map3D map,         /* Evidence grid */
              double height,  /* z-coord of plane to display */
              int x_origin,   /* World x-coord of origin (1/10 inch)*/
              int y_origin)   /* World y-coord of origin (1/10 inch)*/
/*
  Display evidence grid occupancy probabilities in robot window
*/
{
  double xd, yd;              /* Display coords */
  double xscale, yscale, zscale;     /* Cell dimensions (tenths of
inches) */
  double xoffset, yoffset;           /* Circle center offset */
  int x, y, z;                       /* Cell index */
  int xsize, ysize, zsize;           /* Grid dimensions (# cells) */
  int p;                      /* Occupancy probability */
  int empty;                         /* Empty level flag */
/*  int rad;*/                       /* Radius of cell display */

  printf("Displaying grid at (%d, %d)\n", x_origin, y_origin);

  xsize = map.msize[ 0] ;
  ysize = map.msize[ 1] ;
  zsize = map.msize[ 2] ;

  xscale = (map.himv[ 0]  - map.lomv[ 0] ) * 120.0 / (double) xsize;
  yscale = (map.himv[ 1]  - map.lomv[ 1] ) * 120.0 / (double) ysize;
  zscale = (map.himv[ 2]  - map.lomv[ 2] ) * 120.0 / (double) zsize;

  xoffset = xscale / 2.0;
  yoffset = yscale / 2.0;

  z = (int) ((height + HEIGHT_OFFSET - map.lomv[ 2] ) /
          (map.himv[ 2]  - map.lomv[ 2] ) * zsize);

  for (y = 0; y < ysize; y++) {
    for (x = 0; x < xsize; x++) {
      p = map.mapm[ z * xsize * ysize + y * xsize + x] ;
```

```
      xd = (int) ((double) x * xscale + map.lomv[ 0] * 120.0) + x_origin;
      yd = (int) ((double) y * yscale + map.lomv[ 1] * 120.0) + y_origin;

/*      rad = (int) (((double) (p - NEG) / (double) POS) * (double)
xscale * 0.5);

      draw_arc(xd, yd, rad, rad, 0, 3600, 1);*/

      if (p > 0) {
        draw_arc(xd, yd, xscale, yscale, 0, 3600, 1);
      }
      else if (p == 0) {
        draw_arc(xd, yd, xscale / 4.0, xscale / 4.0, 0, 3600, 1);
      }
    }
  }
}

void grid_display_pos(Map3D map,          /* Evidence grid */
                   double height)     /* z-coord of plane to display */
/*
  Display evidence grid occupancy probabilities in robot window at
position
*/
{
  int dx, dy;

  printf("Enter display coordinates ==> ");
  scanf(" %d %d", &dx, &dy);

  grid_display(map, height, dx, dy);
}

void model_init(CylSensorModelArray *sonar_smd,
            CylSensorModelArray *sonar_clear_smd)
/*
  Initialize sensor models
*/
{
  InitCylModelParams();

  MakeCylModel(66, 0.02, 64, 128, 1.0, 22.0, sonar_smd);
  TrimCylModel(*sonar_smd);

/*  WriteCylModel(*sonar_smd, "sonar.mod");*/
/*  ReadCylModel("sonar.mod", sonar_smd);*/

  MakeClearCylModel(66, 0.02, 64, 128, 1.0, 22.0, sonar_clear_smd);
  TrimCylModel(*sonar_clear_smd);

/*  WriteCylModel(*sonar_clear_smd, "clear.mod");*/
/*  ReadCylModel("clear.mod", sonar_clear_smd);*/
}

void sonar_scan(CylSensorModelArray smd, CylSensorModelArray clear_smd,
            Map3D map, int rx, int ry, int rtheta)
/*
```

```
  Update evidence grid using all sonar sensors
*/
{
  PosData sonar_pose[16];      /* Sonar pose information */
  double robot_x, robot_y;     /* Robot position */
  double robot_theta;          /* Robot heading */
  double range;                /* Range reading (feet) */
  double angle;                /* Sensor angle (radians) */
  double sonar_pos[3];         /* Sonar position */
  double sonar_dir[3];         /* Sonar direction */
  int reading;                 /* Raw sonar reading */
  int i;                 /* Sonar index */

  gs();  /* SCOUT THESIS CHANGE use gs to get sonar and position info */

/*  posSonarRingGet(sonar_pose);  SCOUT THESIS CHANGE - comment this
line out  */
/*  SCOUT does not currently provide pose data as NOMAD 200 does  */

  for (i = 0; i < 16; i++) {
/* SCOUT THESIS CHANGE
comment out the requests for pose information below
robot_x = (double) sonar_pose[i].config.configX / 120.0;  commented out
robot_y = (double) sonar_pose[i].config.configY / 120.0;  commented out
robot_theta = (double) sonar_pose[i].config.configTurret / 10.0;
commented out
*/

/* SCOUT THESIS CHANGE uncomment out the lines below
and get the sonar data from using the gs command */
    robot_x = (double) rx / 120.0;
    robot_y = (double) ry / 120.0;
    robot_theta = (double) rtheta / 10.0;

    reading = State[i + 17];
    range = (double) reading / 12.0;
    angle = ((double) i * SONAR_SEP + robot_theta) * M_DEG2RAD;

    sonar_dir[0] = cos(angle);
    sonar_dir[1] = sin(angle);
    sonar_dir[2] = 0.0;

    sonar_pos[0] = sonar_dir[0] * SONAR_RAD;
    sonar_pos[1] = sonar_dir[1] * SONAR_RAD;
    sonar_pos[2] = SONAR_HEIGHT + HEIGHT_OFFSET;

    if ((reading != MAX_SONAR_READING) && (range <= MAX_SONAR_RANGE)){
      AddCylReading(range, sonar_pos, sonar_dir, smd, map);
    }
    else {
      AddCylReading(MAX_SONAR_RANGE, sonar_pos, sonar_dir, clear_smd,
map);
    }
  }
}

void sonar_scan_abs(CylSensorModelArray smd, CylSensorModelArray
clear_smd,
```

```
                 Map3D map, int rx, int ry, int rtheta)
/*
  Update evidence grid using all sonar sensors (using absolute position)
*/
{
  PosData sonar_pose[16];      /* Sonar pose information */
  double robot_x, robot_y;     /* Robot position */
  double robot_theta;          /* Robot heading */
  double range;                /* Range reading (feet) */
  double angle;                /* Sensor angle (radians) */
  double sonar_pos[3];         /* Sonar position */
  double sonar_dir[3];         /* Sonar direction */
  int reading;                 /* Raw sonar reading */
  int i;                  /* Sonar index */

  gs();
/*  posSonarRingGet(sonar_pose);  SCOUT THESIS CHANGE - comment this
line out */
/* SCOUT does not currently provide for pose data as the NOMAD 200 does
*/

  for (i = 0; i < 16; i++) {
/*    robot_x = (double) sonar_pose[i].config.configX / 120.0;  **
comment this line out */
/*    robot_y = (double) sonar_pose[i].config.configY / 120.0;   **
comment out  */
/*    robot_theta = (double) sonar_pose[i].config.configTurret / 10.0;
**  comment out */

/* uncomment out the lines below and use gs command to get sonar data
*/
    robot_x = (double) rx / 120.0;
    robot_y = (double) ry / 120.0;
    robot_theta = (double) rtheta / 10.0;

    reading = State[i + 17];
    range = (double) reading / 12.0;
    angle = ((double) i * SONAR_SEP + robot_theta) * M_DEG2RAD;

    sonar_dir[0]  = cos(angle);
    sonar_dir[1]  = sin(angle);
    sonar_dir[2]  = 0.0;

    sonar_pos[0]  = sonar_dir[0]  * SONAR_RAD + robot_x;
    sonar_pos[1]  = sonar_dir[1]  * SONAR_RAD + robot_y;
    sonar_pos[2]  = SONAR_HEIGHT + HEIGHT_OFFSET;

    if ((reading != MAX_SONAR_READING) && (range <= MAX_SONAR_RANGE)){
      AddCylReading(range, sonar_pos, sonar_dir, smd, map);
    }
    else {
      AddCylReading(MAX_SONAR_RANGE, sonar_pos, sonar_dir, clear_smd,
map);
    }
  }
}
```

```
/* BEGIN SCOUT CHANGE */
/* NOTE - it appears that the following function is never used by any
exploration routine  */
/* Left in code for now  since it will not affect the Scout  */
/* The header for this is in grid++.h  */

void ir_scan_abs(CylSensorModelArray smd, CylSensorModelArray clear_smd,
                 Map3D map, int rx, int ry, int rtheta)
/*
  Update evidence grid using all infrared sensors (using absolute
position)
*/
{
  PosData ir_pose[16];          /* IR pose information */
  double robot_x, robot_y;      /* Robot position */
  double robot_theta;           /* Robot heading */
  double range;                 /* Range reading (feet) */
  double angle;                 /* Sensor angle (radians) */
  double ir_pos[3];             /* IR position */
  double ir_dir[3];             /* IR direction */
  int reading;                  /* Raw IR reading */
  int i;                  /* Sonar index */

  gs();
  posInfraredRingGet(ir_pose);

  for (i = 0; i < 16; i++) {
    robot_x = (double) ir_pose[i].config.configX / 120.0;
    robot_y = (double) ir_pose[i].config.configY / 120.0;
    robot_theta = (double) ir_pose[i].config.configTurret / 10.0;

    /*    robot_x = (double) rx / 120.0;
    robot_y = (double) ry / 120.0;
    robot_theta = (double) rtheta / 120.0;*/

    reading = State[i + 17];
    range = (double) reading / 12.0;
    angle = ((double) i * IR_SEP + robot_theta) * M_DEG2RAD;

    ir_dir[0] = cos(angle);
    ir_dir[1] = sin(angle);
    ir_dir[2] = 0.0;

    ir_pos[0] = ir_dir[0] * IR_RAD + robot_x;
    ir_pos[1] = ir_dir[1] * IR_RAD + robot_y;
    ir_pos[2] = IR_HEIGHT + HEIGHT_OFFSET;

    if (reading < MAX_IR_READING) {
      AddCylReading(range, ir_pos, ir_dir, smd, map);
    }
  }
}

/* END SCOUT CHANGE */


void sonar_scan_abs_norep(CylSensorModelArray smd,
```

```
                        CylSensorModelArray clear_smd,
                        Map3D map, int rx, int ry, int rtheta)
 /*
   Update evidence grid using all sonar sensors (using absolute position)
   (no updates for repeated positions)
 */
 {
   static long old_x[16], old_y[16]; /* Old robot position */
   static int first_flag = 1;  /* Reset first time function is called */

   PosData sonar_pose[16];            /* Sonar pose information */
   double robot_x, robot_y;     /* Robot position */
   double delta;                /* Change in robot position since last
update */
   double robot_theta;          /* Robot heading */
   double range;                /* Range reading (feet) */
   double angle;                /* Sensor angle (radians) */
   double sonar_pos[3];         /* Sonar position */
   double sonar_dir[3];         /* Sonar direction */
   int reading;                 /* Raw sonar reading */
   int i;                 /* Sonar index */

   gs();
   posSonarRingGet(sonar_pose);

   for (i = 0; i < 16; i++) {
     robot_x = (double) sonar_pose[i].config.configX / 120.0;
     robot_y = (double) sonar_pose[i].config.configY / 120.0;
     robot_theta = (double) sonar_pose[i].config.configTurret / 10.0;

     delta = hypot((double) (sonar_pose[i].config.configX - old_x[i]),
                (double) (sonar_pose[i].config.configY - old_y[i]));

     if (first_flag || delta >= MIN_DELTA) {
       old_x[i] = sonar_pose[i].config.configX;
       old_y[i] = sonar_pose[i].config.configY;

       reading = State[i + 17];
       range = (double) reading / 12.0;
       angle = ((double) i * SONAR_SEP + robot_theta) * M_DEG2RAD;

       sonar_dir[0] = cos(angle);
       sonar_dir[1] = sin(angle);
       sonar_dir[2] = 0.0;

       sonar_pos[0] = sonar_dir[0] * SONAR_RAD + robot_x;
       sonar_pos[1] = sonar_dir[1] * SONAR_RAD + robot_y;
       sonar_pos[2] = SONAR_HEIGHT + HEIGHT_OFFSET;

       if ((reading != MAX_SONAR_READING) && (range <= MAX_SONAR_RANGE)){
       AddCylReading(range, sonar_pos, sonar_dir, smd, map);
       }
       else {
       AddCylReading(MAX_SONAR_RANGE, sonar_pos, sonar_dir, clear_smd,
map);
       }
     }
     /*    else {
```

```
        printf("sonar_scan_abs_norep: Repeated position (%d, %d) for
sensor %d.\n",
        old_x[ i] , old_y[ i] , i);
    }*/
  }

  first_flag = 0;
}

void laser_update(Map3D map, double rx, double ry, double lx, double ly,
              double rtheta)
/*
  Update evidence grid for a single laser reading
 */
{
    double lr, ltheta;                /* Laser vector */
    double wx, wy;                    /* World coords of laser endpoint */
    double xsize, ysize;              /* Size of grid cell */
    double stepsize;                  /* Stepsize along laser axis */
    double dx, dy;                    /* Stepsize along x and y axes */
    double px, py;                    /* Point currently being updated */
    int steps;                        /* Number of steps */
    int i;

    lr = hypot(lx, ly);
    ltheta = atan2(ly, lx) * M_RAD2DEG;

    wx = rx + lr * cos((ltheta + rtheta) * M_DEG2RAD);
    wy = ry + lr * sin((ltheta + rtheta) * M_DEG2RAD);

    set_location(map, wx, wy, LASER_HEIGHT, POS);

    px = rx;
    py = ry;

    xsize = (map.himv[ 0]  - map.lomv[ 0] ) / map.msize[ 0] ;
    ysize = (map.himv[ 1]  - map.lomv[ 1] ) / map.msize[ 1] ;

    if (xsize < ysize) {
      stepsize = xsize;
    }
    else {
      stepsize = ysize;
    }

    dx = stepsize * cos((ltheta + rtheta) * M_DEG2RAD);
    dy = stepsize * sin((ltheta + rtheta) * M_DEG2RAD);

    steps = (int) (lr / stepsize);

    for (i = 0; i < steps; i++) {
      set_location(map, px, py, LASER_HEIGHT, NEG);
      px += dx;
      py += dy;
    }
}

void laser_scan(Map3D map, int rx, int ry, int rtheta)
```

```
/*
  Update evidence grid using laser scanner
*/
{
  PosData laser_pose;          /* Laser pose information */
  double lx, ly, lr, ltheta;  /* Laser point (robot coordinates) */
  double wx, wy, wz;          /* Laser point (world coordinates) */
  double robot_x, robot_y, robot_theta;   /* Robot location */
  int i;

  gs();
  posLaserGet(&laser_pose);

  robot_x = (double) laser_pose.config.configX / 120.0;
  robot_y = (double) laser_pose.config.configY / 120.0;
  robot_theta = (double) laser_pose.config.configTurret / 10.0;

  for (i = 0; i < Laser[0]; i++) {
    /*      printf("[%d, %d] ", Laser[i * 2 + 1], Laser[i * 2 + 2]);*/
      if (Laser[i * 2 + 1] != 65000) {
        lx = (double) Laser[i * 2 + 1] / 120.0;
        ly = (double) Laser[i * 2 + 2] / 120.0;
        /*    printf("(%f, %f)", lx, ly);*/

/*      laser_update(map, robot_x, robot_y, lx, ly, robot_theta);*/

        lr = hypot(lx, ly);
        ltheta = atan2(ly, lx) * M_RAD2DEG;

        if (lr <= MAX_LASER_RANGE) {
          wx = lr * cos((ltheta + robot_theta) * M_DEG2RAD);
          wy = lr * sin((ltheta + robot_theta) * M_DEG2RAD);
          wz = LASER_HEIGHT + HEIGHT_OFFSET;
          set_location(map, wx, wy, wz, POS);
          /*          draw_line((int) robot_x * 120.0, (int) robot_y *
120.0, (int) wx * 120.0,
                   (int) wy * 120.0, 19);*/
        }
      }
/*      printf("\n");*/
  }
}

void laser_scan_abs(Map3D map, int rx, int ry, int rtheta)
/*
  Update evidence grid using laser scanner (using absolute position)
*/
{
  PosData laser_pose;          /* Laser pose information */
  double lx, ly, lr, ltheta;  /* Laser point (robot coordinates) */
  double wx, wy, wz;          /* Laser point (world coordinates) */
  double robot_x, robot_y, robot_theta;   /* Robot location */
  int i;

  gs();
  posLaserGet(&laser_pose);

  robot_x = (double) laser_pose.config.configX / 120.0;
```

```
  robot_y = (double) laser_pose.config.configY / 120.0;
  robot_theta = (double) laser_pose.config.configTurret / 10.0;

  for (i = 0; i < Laser[ 0] ; i++) {
    /*      printf("[ %d, %d] ", Laser[ i * 2 + 1] , Laser[ i * 2 + 2] );*/
      if (Laser[ i * 2 + 1] != 65000) {
        lx = (double) Laser[ i * 2 + 1] / 120.0;
        ly = (double) Laser[ i * 2 + 2] / 120.0;
        /*    printf("(%f, %f)", lx, ly);*/

/*      laser_update(map, robot_x, robot_y, lx, ly, robot_theta);*/

        lr = hypot(lx, ly);
        ltheta = atan2(ly, lx) * M_RAD2DEG;

        if (lr <= MAX_LASER_RANGE) {
          wx = lr * cos((ltheta + robot_theta) * M_DEG2RAD) + robot_x;
          wy = lr * sin((ltheta + robot_theta) * M_DEG2RAD) + robot_y;
          wz = LASER_HEIGHT + HEIGHT_OFFSET;
          set_location(map, wx, wy, wz, POS);
          /*          draw_line((int) robot_x * 120.0, (int) robot_y *
120.0, (int) wx * 120.0,
                   (int) wy * 120.0, 19);*/
        }
      }
/*      printf("\n");*/
  }
}

double laser_min(void)
/*
  Return minimum laser range reading
*/
{
  double min_range = MAX_LASER_RANGE;      /* Minimum range reading */
  double lx, ly, lr, ltheta;        /* Laser point (robot coordinates)
*/
  int i;

  gs();

  for (i = 0; i < Laser[ 0] ; i++) {
    if (Laser[ i * 2 + 1] != 65000) {
      lx = (double) Laser[ i * 2 + 1] / 120.0;
      ly = (double) Laser[ i * 2 + 2] / 120.0;

      lr = hypot(lx, ly);

      if (lr < min_range) {
      min_range = lr;
      }
    }
  }

  return(min_range);
}

void lls_scan(CylSensorModelArray smd, CylSensorModelArray clear_smd,
```

```
             Map3D map, int rx, int ry, int rtheta)
/*
  Update evidence grid using laser-limited sonar
*/

{
  PosData sonar_pose[16];       /* Sonar pose information */
  PosData ir_pose[16];          /* IR pose information */
  PosData laser_pose;           /* Laser pose information */
  double sonar_x, sonar_y;      /* Sonar position */
  double sonar_theta;           /* Sonar angle */
  double laser_x, laser_y;      /* Laser position */
  double laser_theta;           /* Laser angle */
  double lx, ly, lr, ltheta;    /* Laser point (robot coordinates) */
  double wx, wy, wz;            /* Laser point (world coordinates) */
  double min_laser_range = MAX_LASER_RANGE;     /* Minimum laser reading
*/
  double sonar_range;           /* Range reading (feet) */
  double angle;                 /* Sensor angle (radians) */
  double sonar_pos[3];          /* Sonar position */
  double sonar_dir[3];          /* Sonar direction */
  double angle_diff;            /* Angle offset between laser and sonar */
  int reading;                  /* Raw sonar reading */
  int i;

  /* Get sensor and pose data from robot */

  gs();
  posSonarRingGet(sonar_pose);
  posInfraredRingGet(ir_pose);
  posLaserGet(&laser_pose);

  /* Update grid using laser readings */

  laser_x = (double) laser_pose.config.configX / 120.0;
  laser_y = (double) laser_pose.config.configY / 120.0;
  laser_theta = (double) laser_pose.config.configTurret / 10.0;

  for (i = 0; i < Laser[0]; i++) {
    /*      printf("[%d, %d] ", Laser[i * 2 + 1], Laser[i * 2 + 2]);*/
      if (Laser[i * 2 + 1] != 65000) {
        lx = (double) Laser[i * 2 + 1] / 120.0;
        ly = (double) Laser[i * 2 + 2] / 120.0;
        /*    printf("(%f, %f)", lx, ly);*/

/*      laser_update(map, laser_x, laser_y, lx, ly, laser_theta);*/

        lr = hypot(lx, ly);
        if (lr < min_laser_range) {
          min_laser_range = lr;
        }

        ltheta = atan2(ly, lx) * M_RAD2DEG;

        if (lr <= MAX_LASER_RANGE) {
          wx = lr * cos((ltheta + laser_theta) * M_DEG2RAD);
          wy = lr * sin((ltheta + laser_theta) * M_DEG2RAD);
          wz = LASER_HEIGHT + HEIGHT_OFFSET;
```

```
          set_location(map, wx, wy, wz, POS);
          /*          draw_line((int) laser_x * 120.0, (int) laser_y *
120.0, (int) wx * 120.0,
                 (int) wy * 120.0, 19);*/
        }
      }
/*      printf("\n");*/
  }

  /* Update grid using sonar reading (limited by minimum laser range) */

  sonar_x = (double) sonar_pose[ 0] .config.configX / 120.0;
  sonar_y = (double) sonar_pose[ 0] .config.configY / 120.0;
  sonar_theta = (double) sonar_pose[ 0] .config.configTurret / 10.0;

  reading = State[ 17] ;

  /* At very close ranges, use infrared instead */

  /*  if (State[ 1]  < MAX_IR_READING) {
    reading = State[ 1] ;

    sonar_x = (double) ir_pose[ 0] .config.configX / 120.0;
    sonar_y = (double) ir_pose[ 0] .config.configY / 120.0;
    sonar_theta = (double) ir_pose[ 0] .config.configTurret / 10.0;

    printf("laser/IR offset = %f inches : %f degrees\n",
    hypot(sonar_x - laser_x, sonar_y - laser_y),
    sonar_theta - laser_theta);
    }
    else {
    printf("laser/sonar offset = %f inches : %f degrees\n",
    hypot(sonar_x - laser_x, sonar_y - laser_y),
    sonar_theta - laser_theta);
  }*/

  /* Compute angle offset between laser and sonar (or IR) */

  angle_diff = fabs(sonar_theta - laser_theta);
  if (angle_diff > 180.0) {
    angle_diff = 360.0 - angle_diff;
  }

  /* Discard reading if offset is too large */

  if (angle_diff > LLS_MAX_ANGLE_DIFF) {
    printf("LLS reading discarded: angle offset = %f\n", angle_diff);
    return;
  }

  /* Determine LLS range */

  sonar_range = (double) reading / 12.0;

  if (sonar_range > min_laser_range) {
    sonar_range = min_laser_range;
  }
```

```
   /* Update grid */

   angle = sonar_theta * M_DEG2RAD;

   sonar_dir[ 0] = cos(angle);
   sonar_dir[ 1] = sin(angle);
   sonar_dir[ 2] = 0.0;

   sonar_pos[ 0] = sonar_dir[ 0] * SONAR_RAD;
   sonar_pos[ 1] = sonar_dir[ 1] * SONAR_RAD;
   sonar_pos[ 2] = SONAR_HEIGHT + HEIGHT_OFFSET;

   if ((reading != MAX_SONAR_READING) && (sonar_range <=
MAX_SONAR_RANGE)){
     if (sonar_range <= MAX_SONAR_OCC_RANGE) {
       AddCylReading(sonar_range, sonar_pos, sonar_dir, smd, map);
     }
     else {
       AddCylReading(sonar_range, sonar_pos, sonar_dir, clear_smd, map);
     }
   }
   else {
     AddCylReading(MAX_SONAR_RANGE, sonar_pos, sonar_dir, clear_smd,
map);
   }
}

void lls_scan_abs(CylSensorModelArray smd, CylSensorModelArray
clear_smd,
              Map3D map, int rx, int ry, int rtheta)
/*
  Update evidence grid using laser-limited sonar (absolute coordinates)
*/

{
  PosData sonar_pose[ 16];        /* Sonar pose information */
  PosData ir_pose[ 16];           /* IR pose information */
  PosData laser_pose;             /* Laser pose information */
  double sonar_x, sonar_y;        /* Sonar position */
  double sonar_theta;             /* Sonar angle */
  double laser_x, laser_y;        /* Laser position */
  double laser_theta;             /* Laser angle */
  double lx, ly, lr, ltheta;      /* Laser point (robot coordinates) */
  double wx, wy, wz;              /* Laser point (world coordinates) */
  double min_laser_range = MAX_LASER_RANGE;      /* Minimum laser reading
*/
  double sonar_range;             /* Range reading (feet) */
  double angle;                   /* Sensor angle (radians) */
  double sonar_pos[ 3];           /* Sonar position */
  double sonar_dir[ 3];           /* Sonar direction */
  double angle_diff;              /* Angle offset between laser and sonar */
  int reading;                    /* Raw sonar reading */
  int i;

  /* Get sensor and pose data from robot */

  gs();
  posSonarRingGet(sonar_pose);
```

```
   posInfraredRingGet(ir_pose);
   posLaserGet(&laser_pose);

   /* Update grid using laser readings */

   laser_x = (double) laser_pose.config.configX / 120.0;
   laser_y = (double) laser_pose.config.configY / 120.0;
   laser_theta = (double) laser_pose.config.configTurret / 10.0;

   for (i = 0; i < Laser[ 0] ; i++) {
     /*       printf("[ %d, %d] ", Laser[ i * 2 + 1] , Laser[ i * 2 + 2] );*/
       if (Laser[ i * 2 + 1]  != 65000) {
         lx = (double) Laser[ i * 2 + 1]  / 120.0;
         ly = (double) Laser[ i * 2 + 2]  / 120.0;
         /*    printf("(%f, %f)", lx, ly);*/

/*       laser_update(map, laser_x, laser_y, lx, ly, laser_theta);*/

         lr = hypot(lx, ly);
         if (lr < min_laser_range) {
           min_laser_range = lr;
         }

         ltheta = atan2(ly, lx) * M_RAD2DEG;

         if (lr <= MAX_LASER_RANGE) {
           wx = lr * cos((ltheta + laser_theta) * M_DEG2RAD) + laser_x;
           wy = lr * sin((ltheta + laser_theta) * M_DEG2RAD) + laser_y;
           wz = LASER_HEIGHT + HEIGHT_OFFSET;
           set_location(map, wx, wy, wz, POS);
           /*          draw_line((int) laser_x * 120.0, (int) laser_y *
120.0, (int) wx * 120.0,
                    (int) wy * 120.0, 19);*/
         }
       }
/*       printf("\n");*/
  }

   /* Update grid using sonar reading (limited by minimum laser range) */

   sonar_x = (double) sonar_pose[ 0] .config.configX / 120.0;
   sonar_y = (double) sonar_pose[ 0] .config.configY / 120.0;
   sonar_theta = (double) sonar_pose[ 0] .config.configTurret / 10.0;

   reading = State[ 17] ;

   /* At very close ranges, use infrared instead */

   /*  if (State[ 1]  < MAX_IR_READING) {
     reading = State[ 1] ;

     sonar_x = (double) ir_pose[ 0] .config.configX / 120.0;
     sonar_y = (double) ir_pose[ 0] .config.configY / 120.0;
     sonar_theta = (double) ir_pose[ 0] .config.configTurret / 10.0;

     printf("IR/sonar offset = %f inches : %f degrees\n",
     hypot(sonar_x - laser_x, sonar_y - laser_y),
     sonar_theta - laser_theta);
```

```
  }
  else {
  printf("laser/sonar offset = %f inches : %f degrees\n",
  hypot(sonar_x - laser_x, sonar_y - laser_y),
  sonar_theta - laser_theta);
  }*/

  /* Compute angle offset between laser and sonar (or IR) */

  angle_diff = fabs(sonar_theta - laser_theta);
  if (angle_diff > 180.0) {
    angle_diff = 360.0 - angle_diff;
  }

  /* Discard reading if offset is too large */

  if (angle_diff > LLS_MAX_ANGLE_DIFF) {
    printf("LLS reading discarded: angle offset = %f\n", angle_diff);
    return;
  }

  /* Determine LLS range */

  sonar_range = (double) reading / 12.0;

  if (sonar_range > min_laser_range) {
    sonar_range = min_laser_range;
  }

  /* Update grid */

  angle = sonar_theta * M_DEG2RAD;

  sonar_dir[ 0] = cos(angle);
  sonar_dir[ 1] = sin(angle);
  sonar_dir[ 2] = 0.0;

  sonar_pos[ 0] = sonar_dir[ 0] * SONAR_RAD + sonar_x;
  sonar_pos[ 1] = sonar_dir[ 1] * SONAR_RAD + sonar_y;
  sonar_pos[ 2] = SONAR_HEIGHT + HEIGHT_OFFSET;

  if ((reading != MAX_SONAR_READING) && (sonar_range <=
MAX_SONAR_RANGE)){
    if (sonar_range <= MAX_SONAR_OCC_RANGE) {
      AddCylReading(sonar_range, sonar_pos, sonar_dir, smd, map);
    }
    else {
      AddCylReading(sonar_range, sonar_pos, sonar_dir, clear_smd, map);
    }
  }
  else {
    AddCylReading(MAX_SONAR_RANGE, sonar_pos, sonar_dir, clear_smd,
map);
  }
}

void clear_robot(Map3D map, int rx, int ry)
{
```

```
    /* Set grid cells under robot to be unoccupied */

    double wx, wy;                  /* Robot location (world/feet) */
    int lx, ly, lz, hx, hy, hz;           /* Corners of robot bounding
box */
    int cx, cy, cz;                 /* Grid cell coordinates */


    wx = (double) rx / 120.0;
    wy = (double) ry / 120.0;

    if (world2grid(map, wx - ROBOT_RADIUS, wy - ROBOT_RADIUS, 0.0,
               &lx, &ly, &lz) == -1) {
      printf("clear_robot: robot edge (%f, %f, %f) out of range.\n",
              wx - ROBOT_RADIUS, wy - ROBOT_RADIUS, 0.0);
      return;
    }
    if (world2grid(map, wx + ROBOT_RADIUS, wy + ROBOT_RADIUS,
ROBOT_HEIGHT,
                &hx, &hy, &hz) == -1) {
      printf("clear_robot: robot edge (%f, %f, %f) out of range.\n",
              wx + ROBOT_RADIUS, wy + ROBOT_RADIUS, ROBOT_HEIGHT);
      return;
    }

    for (cx = lx; cx <= hx; cx++) {
      for (cy = ly; cy <= hy; cy++) {
          for (cz = lz; cz <= hz; cz++) {
            set_grid(map, cx, cy, cz, NEG);
          }
      }
    }
}

void grid_clear(Map3D grid)
/*
  Clear current grid;
 */
{
    ClearMap3D(grid);
}


void grid_decay(Map3D grid)
/*
  Decay grid cells towards base probability
 */
{
    int k, km;

    km =
1<<(ILOG2C(grid.msize[0])+ILOG2C(grid.msize[1])+ILOG2C(grid.msize[2]));

    for (k=0; k<km; k++) {
      if (grid.mapm[k] != 0) {
          if (grid.mapm[k] > GRID_DECAY) {
            grid.mapm[k] -= GRID_DECAY;
          }
```

```
          else if (grid.mapm[ k] < -GRID_DECAY) {
            grid.mapm[ k] += GRID_DECAY;
          }
          else {
            grid.mapm[ k] = 0;
          }
      }
    }
}

void grid_translate(Map3D grid1, Map3D grid2, double dx, double dy)
/*
  Translate all cells in <grid1> by <dx, dy> (feet) and store results in
<grid2>
 */
{
    double wx, wy, wz;          /* World coords */
    int x, y, z;          /* Initial grid cell coordinates */
    int trans_index;            /* Transformed cell index */

    printf("Translating by (%f, %f)\n", dx, dy);

    printf("Initial grid\n");
    grid_display(grid1, SONAR_HEIGHT, 0.0, 0.0);
    printf("Hit <return>\n");
    getchar();

    grid_clear(grid2);

    for (x = 0; x < grid1.msize[ 0] ; x++) {
      for (y = 0; y < grid1.msize[ 1] ; y++) {
          for (z = 0; z < grid1.msize[ 2] ; z++) {
            grid2world(grid1, x, y, z, &wx, &wy, &wz);
            trans_index = world2index(grid1, wx + dx, wy + dy, wz);
            if (trans_index != -1) {
                grid2.mapm[ trans_index] =
                  grid1.mapm[ grid2index(grid1, x, y, z)] ;
            }
          }
      }
    }

    printf("Translated grid\n");
    grid_display(grid2, SONAR_HEIGHT, 0.0, 0.0);
    printf("Hit <return>\n");
    getchar();
}

void grid_rotate(Map3D grid1, Map3D grid2, double dtheta)
/*
  Translate all cells in <grid1> by <dtheta> (degrees) and store results
in
  <grid2>
 */
{
    double wx, wy, wz;          /* Cartesian world coords of initial point
*/
    double rx, ry;              /* Rotated Cartesian coords */
```

```
    double dx, dy;              /* Cartesian vector from center to point
*/
    double radtheta;            /* Rotation in radians */
    double r, theta;            /* Polar coords from center to point */
    int x, y, z;        /* Initial grid cell coordinates */
    int trans_index;            /* Transformed cell index */

    printf("Rotating by (%f)\n", dtheta);

    radtheta = dtheta * M_DEG2RAD;

    printf("Initial grid\n");
    grid_display(grid1, SONAR_HEIGHT, 0.0, 0.0);
    printf("Hit <return>\n");
    getchar();

    grid_clear(grid2);

    for (x = 0; x < grid1.msize[0]; x++) {
      for (y = 0; y < grid1.msize[1]; y++) {
          for (z = 0; z < grid1.msize[2]; z++) {
            grid2world(grid1, x, y, z, &wx, &wy, &wz);

            dx = wx - grid1.cx;
            dy = wy - grid1.cy;

            r = hypot(dy, dx);
            theta = atan2(dy, dx);

            rx = grid1.cx + r * cos(theta + radtheta);
            ry = grid1.cy + r * sin(theta + radtheta);

            trans_index = world2index(grid1, rx, ry, wz);
            if (trans_index != -1) {
                grid2.mapm[trans_index] =
                  grid1.mapm[grid2index(grid1, x, y, z)];
            }
          }
      }
    }

    printf("Rotated grid\n");
    grid_display(grid2, SONAR_HEIGHT, 0.0, 0.0);
    printf("Hit <return>\n");
    getchar();
}


double grid_match(Map3D stm, Map3D local)
/*
  Match two (aligned) evidence grids
 */
{
    double score = 0.0;         /* Match score */
    double wx, wy, wz;          /* World coords of match point */
    int x, y, z;        /* Grid cell coordinates of match point */
    int stm_index;              /* Index of cell in <stm> */
    int p1, p2;                 /* Corresponding probabilities */
```

```
    int m;                /* Match point value (log scale) */
    int sum = 0;          /* Sum of match points values */
    int total = 0;             /* Total # of known points */

    for (x = 0; x < local.msize[0]; x++) {
      for (y = 0; y < local.msize[1]; y++) {
          for (z = 0; z < local.msize[2]; z++) {
            p1 = local.mapm[grid2index(local, x, y, z)];
            grid2world(local, x, y, z, &wx, &wy, &wz);
            stm_index = world2index(stm, wx, wy, wz);
            if (stm_index != -1) {
                p2 = stm.mapm[stm_index];
                total++;
                if (((p1 < 0) && (p2 < 0)) ||
                  ((p1 > 0) && (p2 > 0)) ||
                  ((p1 == 0) && (p2 == 0))) {
                  sum++;
                }
            }
          }
      }
    }

    score = (double) sum / (double) total;
/*    score = (double) sum / (double) (local.msize[0] * local.msize[1] *
                                    local.msize[2]);*/
/*    printf("grid_match: sum = %d : score = %f\n", sum, score);*/

    return(score);
}

double trans_match(Map3D global, Map3D local, double *bx, double *by,
               double *btheta)
/*
  Transform and match two evidence grids
 */
{
    Map3D local_t;             /* Translated local grid */
    Map3D local_tr;            /* Translated/rotated local grid */
    double score;       /* Current match score */
    double best_score;         /* Best match score over all transforms */
    double dx, dy;             /* Current translation distance */
    double dtheta;             /* Current rotation angle */
    double best_x, best_y;     /* Best current translation */
    double best_theta;         /* Best current rotation */
    double vx, vy, vtheta;     /* Transformation vector */
    int sx, sy;                /* Translation step counter */
    int stheta;                /* Rotation step counter */

    grid_init(&local_t, local.cx, local.cy);
    grid_init(&local_tr, local.cx, local.cy);

    vx = 0.0;
    vy = 0.0;
    vtheta = 0.0;

    do {
      best_score = 0.0;
```

```
        for (sx = -NUM_TRANS; sx <= NUM_TRANS; sx++) {
            for (sy = -NUM_TRANS; sy <= NUM_TRANS; sy++) {
              for (stheta = -NUM_ROT; stheta <= NUM_ROT; stheta++) {
                  dx = TRANS_STEP * (double) sx;
                  dy = TRANS_STEP * (double) sy;
                  dtheta = ROT_STEP * (double) stheta;
                  grid_translate(local, local_t, vx + dx, vy + dy);
                  grid_rotate(local_t, local_tr, vtheta + dtheta);
                  score = grid_match(global, local_tr);
                  printf("translation (%f, %f) / rotation (%f): score =
%f\n",
                       dx, dy, dtheta, score);
                  if (score > best_score) {
                    best_score = score;
                    best_x = dx;
                    best_y = dy;
                    best_theta = dtheta;
                  }
              }
            }
        }

        vx += best_x;
        vy += best_y;
        vtheta += best_theta;

        printf("BEST translation (%f, %f) / rotation (%f): score = %f\n",
               best_x, best_y, best_theta, best_score);

      }
      while((best_x != 0.0) || (best_y != 0.0) || (best_theta != 0.0));

      *bx = vx;
      *by = vy;
      *btheta = vtheta;

      return(best_score);
}

void grid_copy(Map3D grid1, Map3D grid2)
/*
  Copy <grid2> to <grid1>
 */
{
      double wx, wy, wz;         /* World coords */
      int x, y, z;         /* Grid cell coordinates */
      int p;               /* Cell occupancy probability */

      for (x = 0; x < grid1.msize[0]; x++) {
        for (y = 0; y < grid1.msize[1]; y++) {
            for (z = 0; z < grid1.msize[2]; z++) {
              grid2world(grid1, x, y, z, &wx, &wy, &wz);
              grid1.mapm[grid2index(grid1, x, y, z)] =
                  grid2.mapm[world2index(grid2, wx, wy, wz)];
            }
        }
      }
```

```
}

void grid_fine_to_coarse(Map3D fine, Map3D coarse)
{
  double wx, wy, wz;             /* World coords */
  int x, y, z;                   /* Grid cell coordinates */
  int p;                   /* Cell occupancy probability */
  int findex;                    /* Index of cell in fine grid */
  int cindex;                    /* Index of cell in coarse grid */
  int cx, cy, cz;

  grid_clear(coarse);

  for (x = 0; x < fine.msize[0]; x++) {
    for (y = 0; y < fine.msize[1]; y++) {
      for (z = 0; z < fine.msize[2]; z++) {
      findex = grid2index(fine, x, y, z);

      grid2world(fine, x, y, z, &wx, &wy, &wz);
      cindex = world2index(coarse, wx, wy, wz);

      world2grid(coarse, wx, wy, wz, &cx, &cy, &cz);

      if (fine.mapm[findex] < 0) {
        coarse.mapm[cindex] -= F2C_CLEAR_WT;
      }

      if (fine.mapm[findex] > 0) {
        coarse.mapm[cindex] += F2C_OCC_WT;
      }

//    if ((coarse.mapm[cindex] == 0) ||
//        (coarse.mapm[cindex] < fine.mapm[findex])) {
//      coarse.mapm[cindex] = fine.mapm[findex];
//    }

      }
    }
  }
}

void integrate_grid(Map3D global,   /* Global grid */
                Map3D local,  /* Local grid */
                double lox,         /* Local x-origin (feet) */
                double loy,         /* Local y-origin (feet) */
                double lotheta)     /* Local origin rotation (degrees)
*/
/*
  Integrate <local> grid within <global> grid
 */
{
  /* Note: Assumes global origin is at (0,0,0) and only handles
     rotations in the xy-plane */

  double lx, ly, lz;           /* Local coords (Cartesian) */
  double lr, ltheta;           /* Local coords (polar) */
  double wx, wy, wz;           /* World coords */
  double ix, iy, itheta;       /* Intermediate coords */
```

```
  int x, y, z;                /* Grid cell coordinates */
  int p;                /* Cell occupancy probability */
  int global_index;           /* Index of global grid cell */
  int local_index;            /* Index of local grid cell */

  printf("integrate_grid: x = %f : y = %f : theta = %f\n", lox, loy,
lotheta);

  lotheta *= M_DEG2RAD;    /* Convert to radians */

  for (x = 0; x < global.msize[0]; x++) {
    for (y = 0; y < global.msize[1]; y++) {
      for (z = 0; z < global.msize[2]; z++) {

      /* Convert cell index to global coords */

      grid2world(global, x, y, z, &wx, &wy, &wz);

      /* Convert global coords to local coords */

      ix = wx - lox;
      iy = wy - loy;
      itheta = atan2(iy, ix);

      lr = hypot(ix, iy);
      ltheta = itheta - lotheta;

      lx = lr * cos(ltheta);
      ly = lr * sin(ltheta);
      lz = wz;           /* Assume z-coord is unchanged */

      /*printf("global (%f, %f) --> local (%f, %f)\n", wx, wy, lx,
ly);*/

      /* Update global cell */

      if ((lx >= X_MIN) && (lx <= X_MAX) &&
          (ly >= Y_MIN) && (ly <= Y_MAX) &&
          (lz >= Z_MIN) && (lz <= Z_MAX)) {
        global_index = grid2index(global, x, y, z);
        local_index = world2index(local, lx, ly, lz);
        global.mapm[global_index] += local.mapm[local_index];

        if (global.mapm[global_index] > POS) {
          global.mapm[global_index] = POS;
        }
        else if (global.mapm[global_index] < NEG) {
          global.mapm[global_index] = NEG;
        }
      }
      }
    }
  }
}

void integrate_global_grid(Map3D global,  /* Initial global grid */
                Map3D global_new,    /* New global grid */
                double nox,          /* Local x-origin (feet) */
```

```
                  double noy,          /* Local y-origin (feet) */
                  double notheta)      /* Local origin rotation (degrees)
*/
/*
  Integrate <local> grid within <global> grid
 */
{
  /* Note: Assumes global origin is at (0,0,0) and only handles
     rotations in the xy-plane */

  double nx, ny, nz;           /* New global coords (Cartesian) */
  double nr, ntheta;           /* New Global coords (polar) */
  double wx, wy, wz;           /* World coords */
  double ix, iy, itheta;       /* Intermediate coords */
  int x, y, z;                 /* Grid cell coordinates */
  int p;                 /* Cell occupancy probability */
  int global_index;            /* Index of global grid cell */
  int new_index;         /* Index of new global grid cell */

  printf("integrate_grid: x = %f : y = %f : theta = %f\n", nox, noy,
notheta);

  notheta *= M_DEG2RAD;   /* Convert to radians */

  for (x = 0; x < global.msize[ 0] ; x++) {
    for (y = 0; y < global.msize[ 1] ; y++) {
      for (z = 0; z < global.msize[ 2] ; z++) {

      /* Convert cell index to global coords */

      grid2world(global, x, y, z, &wx, &wy, &wz);

      /* Convert global coords to new global coords */

      ix = wx - nox;
      iy = wy - noy;
      itheta = atan2(iy, ix);

      nr = hypot(ix, iy);
      ntheta = itheta - notheta;

      nx = nr * cos(ntheta);
      ny = nr * sin(ntheta);
      nz = wz;          /* Assume z-coord is unchanged */

      /*printf("global (%f, %f) --> new global (%f, %f)\n", wx, wy, nx,
ny);*/

      /* Update global cell */

      if ((nx >= GLOBAL_X_MIN) && (nx <= GLOBAL_X_MAX) &&
          (ny >= GLOBAL_Y_MIN) && (ny <= GLOBAL_Y_MAX) &&
          (nz >= GLOBAL_Z_MIN) && (nz <= GLOBAL_Z_MAX)) {
        global_index = grid2index(global, x, y, z);
        new_index = world2index(global_new, nx, ny, nz);
        global.mapm[ global_index]  += global_new.mapm[ new_index] ;

        if (global.mapm[ global_index] > POS) {
```

```
          global.mapm[global_index] = POS;
        }
        else if (global.mapm[global_index] < NEG) {
          global.mapm[global_index] = NEG;
        }
      }
      }
    }
  }
}

void save_grid(Map3D grid)           /* Evidence grid */
/*
  Save evidence grid to file
 */
{
    char filename[80];               /* Save file name */

    printf("Enter save file name ==> ");
    scanf(" %s", filename);

    printf("Saving grid to <%s>.\n", filename);
    WriteMap3D(grid, "Evidence Grid", "", filename);
}

void save_grid_file(Map3D grid,            /* Evidence grid */
                    char *filename,      /* Save file name */
                    char *comment)       /* File header comment */
/*
  Save evidence grid to specified file with header comment
*/
{
  printf("Saving grid to <%s>.\n", filename);
  WriteMap3D(grid, comment, "", filename);
}

void load_grid(Map3D *grid)          /* Evidence grid */
/*
  Load evidence grid from file
 */
{
    char filename[80];               /* Load file name */
    char title[80], footer[80];            /* Discarded */

    printf("Enter load file name ==> ");
    scanf(" %s", filename);

    printf("Loading grid from <%s>.\n", filename);
    if (ReadMap3D(filename, grid, title, footer) == 0) {
      printf("load_grid: Unable to load grid from <%s>.\n", filename);
    }
}

int load_grid_file(Map3D *grid,            /* Evidence grid */
                    char *filename)      /* Load file name */
/*
  Load evidence grid from specified file
```

```
  Returns 1 if successful, 0 otherwise
 */
{
  char title[80], footer[80];       /* Discarded */

  printf("Loading grid from <%s>.\n", filename);
  if (ReadMap3D(filename, grid, title, footer) == 0) {
    printf("load_grid: Unable to load grid from <%s>.\n", filename);
    return (0);
  }
  return(1);
}

int load_grid_file_com(Map3D *grid,       /* Evidence grid */
                       char *filename,        /* Load file name */
                       char *comment)         /* Comment string */
/*
  Load evidence grid from specified file along with header comment

  Returns 1 if successful, 0 otherwise
 */
{
  char footer[80];             /* Discarded */

  printf("Loading grid from <%s>.\n", filename);
  if (ReadMap3D(filename, grid, comment, footer) == 0) {
    printf("load_grid: Unable to load grid from <%s>.\n", filename);
    return (0);
  }
  return(1);
}

void grid_count_occ(Map3D grid, int *occ, int *unocc)
/*
  Count number of occupied and unoccupied cells in grid
*/
{
  int x, y, z;                 /* Grid cell coordinates */
  int xsize, ysize, zsize;     /* Grid dimensions (#cells) */
  int p;                 /* Cell occupancy probability */

  xsize = grid.msize[0];
  ysize = grid.msize[1];
  zsize = grid.msize[2];

  *occ = 0;
  *unocc = 0;

  for (x = 0; x < xsize; x++) {
    for (y = 0; y < ysize; y++) {
      for (z = 0; z < zsize; z++) {
      p = grid.mapm[z * ysize * xsize + y * xsize + x];
      if (p > 0) {
        (*occ)++;
      }
      else if (p < 0) {
        (*unocc)++;
      }
```

```
            }
         }
      }
   }
```

APPENDIX E. FRONTIER-BASED EXPLORATION CODE – ROBOT.H

This appendix contains the header file for the routine that controls many of the robot's basic movement behaviors.

```
/*

  robot.h

  Header file for robot class for Nomad 200 Simulator.
  Original code by Brian Yamauchi

  Modifications for SCOUT THESIS
  By Patrick A. Hillmeyer

*/

#ifndef ROBOT_H

#define ROBOT_H

#include "Nclient.h"
#include "vector.h"
#include "misc.h"
#include "grid++.h"


// BEGIN SCOUT THESIS CHANGE

// These are the conversion macros from Nomadic that accept the steering
and
//  translation values as used for the Nomad 150 and 200 and calculate
the
//  differential-drive axis values for the Nomad Scout.

#define ROTATION_CONSTANT    0.118597  /* inches/degree (known to 100
ppm) */

#define RIGHT(trans, steer)  (trans +
(int)((float)steer*ROTATION_CONSTANT))
#define LEFT(trans, steer)   (trans -
(int)((float)steer*ROTATION_CONSTANT))

#define scout_vm(trans, steer)   vm(RIGHT(trans, steer), LEFT(trans,
steer), 0)
#define scout_pr(trans, steer)   pr(RIGHT(trans, steer), LEFT(trans,
steer), 0)

// END SCOUT THESIS CHANGE



const int NUM_SONAR = 16;     // Number of sensors of each type
const int NUM_IR = 16;     // Actually 0 for SCOUT but leave for now
const int NUM_RANGE = 16;

// BEGIN SCOUT THESIS CHANGE
```

```
const int NUM_TOUCH = 6;    // Scout only has 6 bumper swithes
// END SCOUT THESIS CHANGE

const int NUM_ARC = 16;        // Number of sensor arcs
const int ARC_SIZE = 3;        // Number of sensors in each arc
const int ARC_STEP = 1;        // Interval between first sensor
                        // in each successive arc
const int ARC_OFFSET = -1;     // Value of first sensor of first arc (mod
16)

const int SONAR_ADDR = 17;     // State index for first sonar sensor
const int IR_ADDR = 1;         // State index for first IR sensor
const int TOUCH_ADDR = 33;       // State index for touch sensors
const int LASER_MODE_ADDR = 42; // State index for laser mode

const int MAX_SONAR = 255;     // Maximum sonar reading

// BEGIN SCOUT THESIS CHANGE
const int MAX_IR = 0;          // Maximum IR reading - no IR on Scout
// END SCOUT THESIS CHANGE

const int MAX_RANGE = 255;     // Maximum range reading

const int SENSOR_SEP = 225;    // Separation between adjacent sensors
                        // in degrees/10
// BEGIN SCOUT THESIS CHANGE
// this next setting for BUMPER_SEP can be left as is for now even
// though it is wrong for the Scout because the procedures that use
// it in agent.cc for recoiling from a bumper contact are not
implemented yet
// New NOTE - changed to 600 to fake out some code in robot.cc
// Needs a better fix though
const int BUMPER_SEP = 600;      // Separation between adjacent bumpers
                                 // in degrees/10


const int MAX_SPEED = 200;           // Maximum velocities
const int MAX_TURN_RATE = 300;  // From Nomadic .setup file for Scouts

const int MAX_ACCEL = 300;           // Maximum accelerations
const int MAX_TURN_ACCEL = 500;

const int DEFAULT_SPEED = 200;             // Default velocities
const int DEFAULT_TURN_RATE = 300;  // From Nomadic .setup file for
Scouts

// END SCOUT THESIS CHANGE

const int DEFAULT_ACCEL = 200;             // Default accelerations
const int DEFAULT_TURN_ACCEL = 500;

const int MOVE_TO_SPEED = 10;        // Speed for moving to (x,y)
const int FACE_TURN_RATE = 200;            // Turning rate for facing

const int MAX_CONT_TURN = 225;             // Maximum turn without
stopping
const int FACE_CONT_WAIT = 10;             // How long to wait for turn
to finish
```

```
const int ROBOT_MIN_SPEED = -200;        // Velocity limits (command)
const int ROBOT_MAX_SPEED = 200;
const int ROBOT_MIN_TURN = -100;
const int ROBOT_MAX_TURN = 100;

// BEGIN SCOUT THESIS CHANGE
// NOTE - changing no encoder parameters for now but might need to
//              tweak them for the Scouts

// Dead reckoning parameters

const int ENCODER_COLOR = 19;       // Color of encoder graphic

// Minimum/maximum encoder rotation increment
//const double ENCODER_ROTATE_MIN = 1.0;
//const double ENCODER_ROTATE_MAX = 1.0;
const double ENCODER_ROTATE_MIN = 0.9;
const double ENCODER_ROTATE_MAX = 1.1;

// Encoder rotation bias
const double ENCODER_ROTATE_BIAS = 0.0;
//const double ENCODER_ROTATE_BIAS = 0.001;

// Minimum/maximum encoder translation increment
//const double ENCODER_TRANS_MIN = 1.0;
//const double ENCODER_TRANS_MAX = 1.0;
const double ENCODER_TRANS_MIN = 0.9;
const double ENCODER_TRANS_MAX = 1.1;

// Encoder translation bias
const double ENCODER_TRANS_BIAS = 0.0;
//const double ENCODER_TRANS_BIAS = 0.001;

// Cartesian move parameters

const int MOVE_XY_MAX_DIST = 1200;  // Maximum move (tenths inches)
const int MOVE_XY_MAX_ERROR = 1;    // Maximum move error (tenths
inches)

// END SCOUT THESIS CHANGE

// Building Axis Direction

const int AXIS = 2960;

// Arc directions

enum {  FWD, FFL, FWD_LF, FLL, LF, BLL, BACK_LF, BBL,
        BACK, BBR, BACK_RT, BRR, RT, FRR, FWD_RT, FFR };

// Timeout for movement commands

const unsigned char MOVE_TIMEOUT = 100;

// Robot class definition

class robot {
```

```
 public:
  int id;                         // Robot ID number
  int x, y, theta, turret;          // Robot encoder position
  int actual_x, actual_y, actual_theta;   // Robot actual position
  double enc_x, enc_y, enc_theta;   // Accumulators for encoder position

  int bumper_offset;                // Offset betwen base and bumpers

  double trans_ctr;                 // Total translation since
localization
  double rot_ctr;              // Total rotation since localization

  int origin_x, origin_y;           // Room origin

  int bumpers;                          // Bumper bit vector

  vector sonar;                     // Sensor values
  vector ir;
  vector range;
  vector touch;

  vector abs_sonar;                 // Absolute sensor values
  vector abs_ir;
  vector abs_range;
  vector abs_touch;

  vector arc;                        // Sensor arcs

/*
  vector sonar(NUM_SONAR);          // Sensor values
  vector ir(NUM_IR);
  vector range(NUM_RANGE);
  vector touch(NUM_TOUCH);

  vector abs_sonar(NUM_SONAR);            // Absolute sensor values
  vector abs_ir(NUM_IR);
  vector abs_range(NUM_RANGE);
  vector abs_touch(NUM_TOUCH);

  vector arc(NUM_ARC);         // Sensor arcs
*/

  robot(void);                 // Constructor

  void update(void);           // Sensor update

  void set_default_velocity(void);  // Set default trans/base/turret
speed

  void maint_err(void);       // Maintain encoder error at new position

  // Relative move of <speed>/10 inches forward while turning both
turret
  // and base by <angle>/10 degrees (+ = ccw, - = cw)

  void move(int speed, int angle);

  // Relative move of <speed>/10 inches forward
```

```
void fwd(int speed);

// Turn base by <angle>/10 degrees (+ = ccw, - = cw)

void turn(int angle);

// Set origin to current position

void set_origin_here(void);

// Set origin to (o_x, o_y)

void set_origin_loc(int o_x, int o_y);

// Convert angle to sensor index

int theta2sensor(double theta);

// Convert sensor index to angle

double sensor2theta(int sensor);

// Return 1 if all range sensors in an arc <width> x 2 sensors wide
// centered around sensor <ctr> return greater or equal to <dist>,
// 0 otherwise.

int check_clear(int ctr, int width, int dist);

// Wrap index to [ 0..NUM_SENSORS]

int sensor_wrap(int index);

// Turn off sensors

void shutdown(void);

// Move robot to <x, y> (world coords, tenths of inch)

int move_to_xy(int cx, int cy);

// Turn robot to face <angle> accurately

void face_angle(int angle);

// Turn robot to face <angle> quickly

void face_angle_fast(int angle);

// Turn robot to face <angle> quickly, without stopping

void face_angle_cont(int angle);

// Align turret with base

void turret_align(void);

// Relative Cartesian move to <x, y> (world coords, tenths of inches)
```

```
    void move_rel(int x, int y);

    // Sensor functions

    void sonar_on(void);
    void sonar_single(int index);           // Index of sensor to fire
    void sonar_off(void);
    void ir_on(void);
    void ir_single(int index);        // Index of sensor to fire
    void ir_off(void);
    void laser_on(void);
    void laser_off(void);

    // Wait for the robot to start moving (any motor)

    void wait_start(void);

   private:

    // Initialization functions

    void initialize_sensors(void);

    // Update functions

    void update_dr(void);
    void update_range_arcs(void);
    void update_arc(int &av, int first, int last);

    // Cleanup functions
    void deactivate_sonar(void);


  };

  #endif
```

APPENDIX F. FRONTIER-BASED EXPLORATION CODE – ROBOT.CC

This appendix contains the source code for the routine that controls many of the robot's basic movement behaviors.

```
/*

  robot.cc

  Robot class for Nomad 200 Simulator.
  Original code by Brian Yamauchi

  Modifications for SCOUT THESIS
  By Patrick A. Hillmeyer

*/

#include <iostream.h>
#include <math.h>

#include "robot.h"
#include "drand.h"
#include "irand.h"

// Dead reckoning mode (actual, independent, or error)

#define DR_ACTUAL

// Touch vector mask

const int touch_mask[ 20] = { 1, 2, 4, 8, 16,
                       32, 64, 128, 256, 512,
                       1024, 2048, 4096, 8192, 16384,
                       32768, 65536, 131072, 262144, 524288 };

// Forward contact mask

const int FWD_CONTACT = 1015839;

robot::robot(void):sonar(NUM_SONAR), ir(NUM_IR), range(NUM_RANGE),
                   touch(NUM_TOUCH), abs_sonar(NUM_SONAR),
abs_ir(NUM_IR),
                   abs_range(NUM_RANGE), abs_touch(NUM_TOUCH),
arc(NUM_ARC)
{
  int rx, ry, rtheta;         // Robot home position (1/10 inch, 1/10
degree)

  // Connect to server and activate all sensors

  cout << "Enter Nserver host name ==> ";
  cin >> SERVER_MACHINE_NAME;

  cout << "Enter Nserver robot ID ==> ";
  cin >> id;

  connect_robot(id);
```

```
  initialize_sensors();
  set_default_velocity();

  // Initialize origin

  origin_x = 0;
  origin_y = 0;

  // Initialize translation/rotation counters

  trans_ctr = 0.0;
  rot_ctr = 0.0;

  // Zero the robot

  //  zr();

  // Set robot initial position

  //  tk("Align me.");

  cout << "Enter robot x y theta (no commas) ==> ";
  cin >> rx >> ry >> rtheta;

  gs();
  bumper_offset = State[36] - rtheta;

  place_robot(rx, ry, rtheta, rtheta);

  // Initialize actual position

  gs();
  actual_x = State[34];
  actual_y = State[35];
  actual_theta = State[36];   // DR heading
//  actual_theta = 3600 - wrap(State[43] - AXIS, 0, 3600);  // Compass
heading

  // Initialize encoder accumulators

  enc_x = (double) actual_x;
  enc_y = (double) actual_y;
  enc_theta = (double) actual_theta;

  // Initialize estimated position

  x = round(enc_x);
  y = round(enc_y);
  theta = round(enc_theta);

  // Display robot estimated position

//  draw_robot(x, y, theta, theta, ENCODER_COLOR);

  // Updater robot state

  update();
}
```

```
void robot::maint_err(void)
{
    // Maintain encoder error at new position

    int ex, ey, etheta;

    ex = x - actual_x;
    ey = y - actual_y;
    etheta = theta - actual_theta;

    gs();
    place_robot(State[ 34] , State[ 35] , State[ 36] , State[ 37] );

    actual_x = State[ 34] ;
    actual_y = State[ 35] ;
    actual_theta = State[ 36] ;

    x = actual_x + ex;
    y = actual_y + ey;
    theta = actual_theta + etheta;
}

void robot::set_default_velocity()
{
  sp(DEFAULT_SPEED, DEFAULT_TURN_RATE, 0);    // TEMP FIX for SCOUT
  ac(DEFAULT_ACCEL, DEFAULT_TURN_ACCEL, 0); // TEMP FIX for SCOUT
}

void robot::update(void)
{
  // Update values for position <x, y, theta>, sonar <sonar>, infrared
  // sensors <ir>.  Also update range arcs.

  int range_offset;             // Rotation offset for range sensors
  int touch_offset;             // Rotation offset for touch sensors
  int i;

  gs();

  update_dr();

  range_offset = (int) ((double) theta / (double) SENSOR_SEP + 0.5);

// NOTE - need to fix this BUMPER_SEP dependency later for SCOUT

  touch_offset = wrap((int) ((double) (theta + bumper_offset)
                     / (double) BUMPER_SEP + 0.5),
                 NUM_TOUCH);

  //  cout << "Offset = " << range_offset << "" << endl;

  for (i = 0; i < NUM_SONAR; i++) {
    sonar[ i] = State[ i + SONAR_ADDR] ;
    abs_sonar[ i] = State[ wrap(i - range_offset, NUM_SONAR) +
SONAR_ADDR] ;
```

```
    //    cout << "i = " << i << " : range_offset i = " << wrap(i -
range_offset, NUM_SONAR) <<
    //       " : sonar[" << i << "] = " << sonar[i] << "" << endl;
  }

// BEGIN SCOUT THESIS CHANGE
// Comment out IR code and only depend on sonars

// SCOUT THESIS CHANGE -  correct error where sensor updates below were
left out of loop
  for (i = 0; i < NUM_SONAR; i++) {
   abs_range[i] = abs_sonar [i];
   range[i] = sonar[i];
// cout << "Just set by sonar[i] value : range[" << i << "] = " <<
range[i]
//        << endl;       // TEMP FIX
// cout << "Just set by abs_sonar[i] value : abs_range[" << i << "] = "
//        << abs_range[i] << endl;      // TEMP FIX

  }  // end for loop

//  for (i = 0; i < NUM_IR; i++) {
//    ir[i] = State[i + IR_ADDR];
//    abs_ir[i] = State[wrap(i - range_offset, NUM_IR) + IR_ADDR];

    //    cout << "i = " << i << " : range_offset i = " << wrap(i -
range_offset, NUM_IR) <<
    //       " : ir[" << i << "] = " << ir[i] << "" << endl;
//  }

//  for (i = 0; i < NUM_RANGE; i++) {
//    if (abs_ir[i] < abs_sonar[i]) {
//      abs_range[i] = abs_ir[i];
//    }
//    else {
//      abs_range[i] = abs_sonar[i];
//    }
//
//    if (ir[i] < sonar[i]) {
//      range[i] = ir[i];
//    }
//    else {
//      range[i] = sonar[i];
//    }
//  }

// END SCOUT THESIS CHANGE

  for (i = 0; i < NUM_RANGE; i++) {
    if (range[i] > MAX_RANGE) {
      range[i] = MAX_RANGE;
// cout << "Compared against MAX_RANGE : range[" << i << "] = " <<
range[i]
//       << endl;     // TEMP FIX
    }
    if (abs_range[i] > MAX_RANGE) {
      abs_range[i] = MAX_RANGE;
```

```
// cout << "Compared against MAX_RANGE : abs_range[" << i << "] = " <<
abs_range[ i]
//          << endl;    // TEMP FIX
    }
  }    // end for loop

  update_range_arcs();

  bumpers = State[ TOUCH_ADDR] ;
  //  if (bumpers != 0) {
  //      cout << "Bumper state = " << bumpers << "" << endl;
  //  }

// NOTE - touch_offset depends on BUMPER_SET - needs fix for SCOUT

  for (i = 0; i < NUM_TOUCH; i++) {
    if (bumpers &
      touch_mask[ wrap(i + touch_offset, NUM_TOUCH)] ) {
      touch[ i] = 1;
      cout << "Contact on bumper " << i << " (abs index = " <<
      wrap(i + touch_offset, NUM_TOUCH) << ")" << endl;
    }
    else {
      touch[ i] = 0;
    }
  }

// cout << "Sonar = " << sonar << endl;    // TEMP FIX
  //  cout << "IR = " << ir << endl;
// cout << "Range = " << range << endl;     // TEMP FIX
// cout << "Arcs = " << arc << endl;     // TEMP FIX
  //  cout << "Touch = " << touch << endl;
}

void robot::update_dr(void)
{
  // Update dead reckoning

  double dx, dy, dtheta;              // Motion since last update
  int dtheta_mag;                // Magnitude of rotation
  int dtheta_sgn;                // Direction of rotation (+ ccw, - cw)

  double vec_r, vec_theta;            // Motion vector
  double inc;                         // Motion increment
  double ctheta, stheta;              // Components along x-axis and y-
axis
  double trans_step;                  // Length of current translation
step

  int i;

  dx = (double) (State[ 34] - actual_x);
  dy = (double) (State[ 35] - actual_y);
  dtheta = angle_sgn_diff((double) State[ 36] / 10.0,
                    (double) actual_theta / 10.0) * 10.0;

  actual_x = State[ 34] ;
  actual_y = State[ 35] ;
```

```
  actual_theta = State[ 36] ;   // DR heading
//  actual_theta = 3600 - wrap(State[ 43] - AXIS, 0, 3600);  // Compass
heading

#ifdef DR_ACTUAL

  // Dead reckoning always returns actual position

  x = actual_x;
  y = actual_y;
  theta = actual_theta;
// SCOUT THESIS CHANGE - comment out original turret line
// set turret to be same as SCOUT heading angle
//  turret = State[ 37] ;
   turret = theta;

#endif // DR_ACTUAL

#ifdef DR_INDEP

  // Dead reckoning is updated by actual displacements, but may be set
  // independently

  draw_robot(x, y, theta, theta, ENCODER_COLOR);

  x += (int) dx;
  y += (int) dy;
  theta += (int) dtheta;

  draw_robot(x, y, theta, theta, ENCODER_COLOR);

#endif // DR_INDEP

#ifdef DR_ERROR

  // Dead reckoning accumulates error over time

  draw_robot(x, y, theta, theta, ENCODER_COLOR);

  rot_ctr += fabs(dtheta);

  dtheta_mag = (int) fabs(dtheta);
  dtheta_sgn = sgn(dtheta);
  for (i = 0; i < dtheta_mag; i++) {
    enc_theta += (double) dtheta_sgn *
      rdrand(ENCODER_ROTATE_MIN, ENCODER_ROTATE_MAX) +
ENCODER_ROTATE_BIAS;
  }
  enc_theta = angle_wrap(enc_theta / 10.0) * 10.0;

  theta = round(enc_theta);

  vec_r = hypot(dx, dy);

  if (vec_r > 0.0) {
    trans_ctr += vec_r;

    vec_theta = (double) theta / 10.0;
```

```
      if (angle_diff(vec_theta, atan2(dy, dx) * RAD2DEG) > 90.0) {
        vec_theta = angle_wrap(vec_theta + 180.0);
      }

      ctheta = cos(vec_theta * DEG2RAD);
      stheta = sin(vec_theta * DEG2RAD);

      for (i = 0; i < (int) vec_r; i++) {
        trans_step = rdrand(ENCODER_TRANS_MIN, ENCODER_TRANS_MAX) +
        ENCODER_TRANS_BIAS;
        trans_step = 1.0;
        enc_x += ctheta * trans_step;
        enc_y += stheta * trans_step;
      }

      enc_x += ctheta * (vec_r - (int) vec_r);
      enc_y += stheta * (vec_r - (int) vec_r);

      x = round(enc_x);
      y = round(enc_y);
    }

    draw_robot(x, y, theta, theta, ENCODER_COLOR);

  /*  cout << "Actual: (" << actual_x << ", " << actual_y << ") <" <<
  actual_theta
      << "> -- Encoder: (" << enc_x << ", " << enc_y << ") <" << enc_theta
  <<
       "> -- Error: (" << enc_x - (double) actual_x << ", " <<
       enc_y - (double) actual_y << ") <" <<
         round(angle_sgn_diff(enc_theta / 10.0,
                  (double) actual_theta / 10.0)
             * 10.0) << ">" << endl;

    cout << "Total: translation = " << trans_ctr << " : rotation = " <<
      rot_ctr << endl;*/

  #endif // DR_ERROR

    return;
  }

  void robot::update_range_arcs(void)
  {
    // Update range arcs.  The value of the arc is equal to the minimum
    // range reading of a sensor that is included in that arc.

    int i, first, last;

    for (i = 0; i < NUM_ARC; i++) {
      first = wrap(i * ARC_STEP + ARC_OFFSET, NUM_RANGE);
      last = wrap(first + ARC_SIZE - 1, NUM_RANGE);

      arc[ i] = range.min(first, last);
    }
  }

  void robot::sonar_on(void)
```

```
{
  // Activate all sonar sensors

    int sn_order[ 16] ;   // Sonar firing order

    /* set firing rate and sequence of all sonar */
    sn_order[ 0]  =  0; sn_order[ 1]  = 10; sn_order[ 2]  =  6;
    sn_order[ 3]  = 14; sn_order[ 4]  =  2; sn_order[ 5]  = 12;
    sn_order[ 6]  =  4; sn_order[ 7]  =  9; sn_order[ 8]  =  1;
    sn_order[ 9]  = 13; sn_order[ 10]  =  5; sn_order[ 11]  = 15;
    sn_order[ 12]  = 7; sn_order[ 13]  =  11; sn_order[ 14]  = 3;
    sn_order[ 15]  = 8;
    conf_sn (10, sn_order);     // TEMP FIX SET LONGER SONAR FIRING TIME
}

void robot::sonar_single(int index) // Index of sensor to fire
{
  // Activate one sonar sensor

  int sn_order[ 16] ;     // Sonar firing order

  sn_order[ 0]  = index;
  sn_order[ 1]  = 255;

  conf_sn (12, sn_order);
}

void robot::sonar_off(void)
{
  // Deactivate all sonar sensors

  int sn_order[ 16] ;     // Sonar firing order

  sn_order[ 0]  = 255;
  conf_sn(1, sn_order);
}


// BEGIN SCOUT THESIS CHANGE
// let the IRs and laser be configured - just comment out the activation
//  in the following procedures

void robot::ir_on(void)
{
  // Activate all IR sensors

    int ir_order[ 16] ;   // IR firing order

   /* set firing rate and sequence of all IR */
    ir_order[ 0]  =  0; ir_order[ 1]  = 10; ir_order[ 2]  =  6;
    ir_order[ 3]  = 14; ir_order[ 4]  =  2; ir_order[ 5]  = 12;
    ir_order[ 6]  =  4; ir_order[ 7]  =  9; ir_order[ 8]  =  1;
    ir_order[ 9]  = 13; ir_order[ 10]  =  5; ir_order[ 11]  = 15;
    ir_order[ 12]  = 7; ir_order[ 13]  =  11; ir_order[ 14]  = 3;
    ir_order[ 15]  = 8;
//    conf_ir (2, ir_order);
}
```

```
void robot::ir_single(int index)    // Index of sensor to fire
{
  // Activate one IR sensor

  int ir_order[16];      // IR firing order

  ir_order[0] = index;
  ir_order[1] = 255;

//  conf_ir(2, ir_order);
}

void robot::ir_off(void)
{
  // Deactivate all IR sensors

  int ir_order[16];      // IR firing order

  ir_order[0] = 255;
//  conf_ir(2, ir_order);
}

void robot::laser_on(void)
{
  // Activate laser

//  conf_ls(LASER_MODE_ON, THRESHHOLD, WIDTH, NUMDATA, AVG);
}

void robot::laser_off(void)
{
  // Deactivate laser

//  conf_ls(LASER_MODE_OFF, THRESHHOLD, WIDTH, NUMDATA, AVG);
}

void robot::initialize_sensors(void)
{
    /*
      Activate all robot sensors
      */
//    static int ir_on[16] ={0, 10, 6, 14, 2, 12, 4, 9, 1, 13, 5, 15, 7,
11, 3, 8};
//    static int ir_off[16]={255, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
13, 14, 15};

    static int sn_on[16] ={0, 10, 6, 14, 2, 12, 4, 9, 1, 13, 5, 15, 7,
11, 3, 8};
    static int sn_off[16]={255, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
13, 14, 15};

    int i;

    /*  init_sensors(); */

//    conf_ir(0, ir_on);
    conf_sn(10, sn_on);  //TEMP FIX - set longer slower firing time

```

```
    Smask[ 42] = 1;
//    conf_ls(LASER_MODE_ON, THRESHHOLD, WIDTH, NUMDATA, AVG);

    /*  conf_cp(1); */  /* =-=-= doesn't work with Ndirect.o */

    posDataRequest(POS_SONAR);          // Just get the sonar data for
the SCOUT
    if (posDataCheck() != (POS_SONAR)) {          // Just check for the
sonar data for the SCOUT
      cout << "\nERROR: Could not set up pose info for sensors.\n";
      exit(-1);
    }
}

// END SCOUT THESIS CHANGE



void robot::move(int speed, int angle)
{
  // Relative move of <speed>/10 inches forward while turning
  // base and turret by <angle>/10 degrees (+ = ccw, - = cw)

  int vel_speed, vel_angle;   // Velocity commands

  vel_speed = limit(speed, ROBOT_MIN_SPEED, ROBOT_MAX_SPEED);
  vel_angle = limit(angle, ROBOT_MIN_TURN, ROBOT_MAX_TURN);

// BEGIN SCOUT THESIS CHANGE
  scout_vm(vel_speed, vel_angle);

//  scout_pr(speed, angle);
}

void robot::fwd(int speed)
{
  // Relative move of <speed>/10 inches forward

// TEMP SCOUT FIX- change pr cmds to vm cmds and comment out the wait
  scout_vm(speed, 0);
//  ws(1, 1, 0, 5);     TEMP FIX - comment out the wait
}

void robot::turn(int angle)
{
  // Turn base and turret by <angle>/10 degrees (+ = ccw, - = cw)

// TEMP FIX - change pr cmds to vm cmds and comment out the wait
    scout_vm(0, angle);
//    ws(1, 1, 0, 5);  TEMP FIX - comment out the wait
}

// END SCOUT THESIS CHANGE


void robot::set_origin_here(void)
{
  // Define current position and orientation as origin
```

```
/*  dp(0, 0);
  da(0, 0);*/

  origin_x = State[ 34] ;
  origin_y = State[ 35] ;

  x = 0;
  y = 0;
  theta = 0;

  cout << "Setting origin to (" << origin_x << ", " << origin_y << ")."
<< endl;
}

void robot::set_origin_loc(int o_x, int o_y)
{
  // Define new origin relative to current origin

  origin_x += o_x;
  origin_y += o_y;

  x = 0;
  y = 0;
  theta = 0;

  cout << "Setting origin to (" << origin_x << ", " << origin_y << ")."
<< endl;
}

int robot::theta2sensor(double theta)
{
    // Convert angle to sensor index

    int sensor;

    sensor = (int) (theta * 10.0) / SENSOR_SEP;
    return(sensor);
}

double robot::sensor2theta(int sensor)
{
    // Convert sensor index to angle

    double theta;

    theta = (double) (sensor * SENSOR_SEP) / 10.0;
    return(theta);
}

int robot::sensor_wrap(int index)
{
    // Wrap index to [ 0..NUM_RANGE]

    int windex;

    windex = wrap(index, NUM_RANGE);
    return(windex);
```

```
}

int robot::check_clear(int ctr, int width, int dist)
{
    // Return 1 if all range sensors in an arc <width> x 2 sensors wide
    // centered around sensor <ctr> return greater or equal to <dist>,
    // 0 otherwise.

    int left, right;
    int min_dist;

    left = sensor_wrap(ctr + width);
    right = sensor_wrap(ctr - width);

    min_dist = range.min(right, left);

    if (min_dist >= dist) {
      return(1);
    }
    else {
      return(0);
    }
}

void robot::shutdown(void)
{
  sonar_off();
  ir_off();
  laser_off();
}

int robot::move_to_xy(int cx, int cy)
{
    // Move robot to <x, y> (world coords, tenths of inch)

    int dx, dy;         // Difference between current and desired
positions
    double dist;  // Distance to destination
    double angle; // Bearing to destination
    double mturn; // Turn command

// BEGIN SCOUT THESIS CHANGE
    scout_vm(0, 0);
// END SCOUT THESIS CHANGE
    sp(MOVE_TO_SPEED, DEFAULT_TURN_RATE, 0);  // TEMP FIX for SCOUT

    update();

    cout << "current position (" << x << ", " << y << ") : destination
("
       << cx << ", " << cy << ")" << endl;

    dx = cx - x;
    dy = cy - y;

    dist = hypot((double) dx, (double) dy);
    if (dist == 0.0) {
      angle = 0.0;
```

```
    }
    else {
      angle = atan2((double) dy, (double) dx) * RAD2DEG;
    }

    cout << "distance = " << dist << " : angle = " << angle << endl;

    while((dist > MOVE_XY_MAX_ERROR) && (touch.max(0, NUM_TOUCH - 1) ==
0)) {
      if (dist > MOVE_XY_MAX_DIST) {
          cout << "Destination too far (" << dist / 10.0 << " inches)"
<<
            endl;
          return(0);
      }

      mturn = (int) (angle_sgn_diff(angle, (double) theta / 10.0) *
10.0);
// BEGIN SCOUT THESIS CHANGE

// TEMP FIX - change pr cmds to vm cmds and commnent out the ws cmds
      scout_vm(0, (int) mturn);
//    ws(1, 1, 0, 100);     TEMP FIX - comment out wait

      scout_vm((int) dist, 0);
//    ws(1, 1, 0, 100);     TEMP FIX - comment out wait

// END SCOUT THESIS CHANGE

      update();

      cout << "current position (" << x << ", " << y << ") : destination
("
           << cx << ", " << cy << ")" << endl;

      dx = cx - x;
      dy = cy - y;

      dist = hypot((double) dx, (double) dy);
      if (dist == 0.0) {
          angle = 0.0;
      }
      else {
          angle = atan2((double) dy, (double) dx) * RAD2DEG;
      }

      cout << "distance = " << dist << " : angle = " << angle << endl;
    }

    set_default_velocity();

    return(1);
}

void robot::face_angle(int angle)   // Desired angle (1/10 degree)
{
  // Turn robot to face <angle> accurately
```

```
  int dtheta;           // Difference between current and desired angle

  cout << "Facing angle <" << angle << ">" << endl;

// BEGIN SCOUT THESIS CHANGE

  scout_vm(0, 0);
  sp(DEFAULT_SPEED, FACE_TURN_RATE, 0);  // TEMP FIX for SCOUT

  update();
  dtheta = (int)
    (angle_sgn_diff((double) angle / 10.0, (double) theta / 10.0) *
10.0);

  while(dtheta != 0) {
    cout << "current angle = " << theta << " : turn = " << dtheta <<
endl;

// TEMP FIX - change below to vm vice pr and comment out the wait
    scout_vm(0, dtheta);
//    ws(1, 1, 0, 100);    TEMP FIX comment out the wait

// END SCOUT THESIS CHANGE

    update();
    dtheta = (int)
      (angle_sgn_diff((double) angle / 10.0, (double) theta / 10.0) *
10.0);
  }

  cout << "Alignment complete." << endl;

  set_default_velocity();
}

void robot::face_angle_fast(int angle)    // Desired angle (1/10 degree)
{
  // Turn robot to face <angle> quickly

  int dtheta;           // Difference between current and desired angle

  dtheta = (int)
    (angle_sgn_diff((double) angle / 10.0, (double) theta / 10.0) *
10.0);

// TEMP FIX for SCOUT to line below
//  cout << "face_angle_fast(" << angle << ") : scout_vm(0, " << dtheta
//       << ")" << endl;     // TEMP FIX for SCOUT

// BEGIN SCOUT THESIS CHANGE

// TEMP FIX - change pr cmds to vm cmds and comment out the wait
  scout_vm(0, dtheta);
//  ws(1, 1, 0, 10);   TEMP FIX - comment out the wait
// END SCOUT THESIS CHANGE
}

void robot::face_angle_cont(int angle)    // Desired angle (1/10 degree)
```

```
{
  // Turn robot to face <angle> quickly, without stopping

  int dtheta;           // Difference between current and desired angle

  dtheta = (int)
    (angle_sgn_diff((double) angle / 10.0, (double) theta / 10.0) *
10.0);

  cout << "face_angle_cont(" << angle << ") : pr(0, " << dtheta << ", "
       << dtheta << ")" << endl;

  if ((dtheta < -MAX_CONT_TURN) || (dtheta > MAX_CONT_TURN)) {
//    mv(MV_VM, 0, MV_PR, dtheta, MV_PR, dtheta);  // TEMP FIX - comment
this line out
    mv(MV_VM, 0, MV_PR, dtheta, MV_IGNORE, 0);  // TEMP FIX - try to fix
SCOUT problem
    ws(1, 1, 0, FACE_CONT_WAIT);    // wait for both wheels to stop for
SCOUT
  }
  else {
//    mv(MV_IGNORE, 0, MV_PR, dtheta, MV_PR, dtheta);  // TEMP FIX -
comment this line out
    mv(MV_IGNORE, 0, MV_PR, dtheta, MV_IGNORE, 0);   // TEMP FIX - try
to fix SCOUT problem
  }
}

// BEGIN SCOUT THESIS CHANGE
// do not need this next routine because there is no separate
// turret to align on the SCOUT
// leave as is because call to turret_align has been commented out
// in agent.cc
void robot::turret_align(void)
{
    // Align turret with base

    int turret;          // Turret angle
    int dtheta;          // Difference between turret and base

    scout_vm(0, 0);
    sp(DEFAULT_SPEED, FACE_TURN_RATE, 0);   // TEMP FIX for SCOUT

    update();
    turret = State[ 37];
    dtheta = 0;     // fake it for SCOUT
//    dtheta = wrap(actual_theta - turret, -1800, 1800);

    while(dtheta != 0) {
      scout_vm(0, 0);    // TEMP TECK for SCOUT
      ws(0, 0, 0, 100);

      update();
      turret = State[ 37];
      dtheta = wrap(actual_theta - turret, -1800, 1800);
    }

// END SCOUT THESIS CHANGE
```

```
    set_default_velocity();
}

void robot::move_rel(int x, int y)
{
    // Relative Cartesian move to <x, y> (world coords, tenths of
inches)

    int old_angle;
    int move_angle;
    int move_dist;

    update();
    old_angle = State[ 36] ;

    move_angle = (int) (atan2((double) y, (double) x) * RAD2DEG * 10.0);
    move_dist = (int) hypot((double) x, (double) y);

    face_angle(move_angle);

    sp(MOVE_TO_SPEED, DEFAULT_TURN_RATE, 0);   // TEMP FIX for SCOUT
// BEGIN SCOUT THESIS CHANGE

// TEMP FIX - change pr to vm and comment out the wait
    scout_vm(move_dist, 0);
//    ws(1, 1, 0, 100);    TEMP FIX - comment out the wait
// END SCOUT THESIS CHANGE
    set_default_velocity();

    face_angle(old_angle);
}

void robot::wait_start(void)
{
  // Wait for the robot to start moving (any motor)

  gs();

  while((State[ STATE_VEL_TRANS] == 0) &&
      (State[ STATE_VEL_STEER] == 0) &&
      (State[ STATE_VEL_TURRET] == 0)) {
    gs();
  }
}
```

APPENDIX G. FRONTIER-BASED EXPLORATION CODE – AGENT.H

This appendix contains the header file for the routine that controls the robot's exploration behaviors.

```
/*

  agent.h

  Header file for agent class

*/

#ifndef AGENT_H
#define AGENT_H

#include "drand.h"
#include "irand.h"
#include "robot.h"
#include "place_net.h"
#include "arb.h"
#include "control_panel.h"
#include "bar_graph.h"
#include "mobstacle.h"
#include "comm++.h"
#include "comm.h"
#include "frontier.h"
#include "path.h"

// BEGIN SCOUT THESIS CHANGE

// These are the conversion macros from Nomadic that accept the steering
and
//  translation values as used for the Nomad 150 and 200 and calculate
the
//  differential-drive axis values for the Nomad Scout.

#define ROTATION_CONSTANT    0.118597  /* inches/degree (known to 100
ppm) */

#define RIGHT(trans, steer)  (trans +
(int)((float)steer*ROTATION_CONSTANT))
#define LEFT(trans, steer)   (trans -
(int)((float)steer*ROTATION_CONSTANT))

#define scout_vm(trans, steer)   vm(RIGHT(trans, steer), LEFT(trans,
steer), 0)
#define scout_pr(trans, steer)   pr(RIGHT(trans, steer), LEFT(trans,
steer), 0)

// END SCOUT THESIS CHANGE



// Motor control parameters

const int SPEED_RES = 40;
```

```
const double SPEED_MIN = -100.0;
const double SPEED_MAX = 100.0;
const double SPEED_DEF = 0.0;
const double SPEED_NOISE = 5.0;

const int TURN_RES = 32;
const double TURN_MIN = -180.0;
const double TURN_MAX = 180.0;
const double TURN_DEF = 0.0;
const double TURN_NOISE = 5.0;

// Random turn to escape stasis

const double RAND_TURN = 10.0;

// Speed arbitrator window parameters

const int SPWIN_X = 570;            // x-coord of top
const int SPWIN_Y = 460;            // y-coord of left side
const int SPWIN_WIDTH = 175;        // Window width
const int SPWIN_HEIGHT = 50;        // Window height
const double SPWIN_MIN = -20.0;          // Minimum vote total
const double SPWIN_MAX = 20.0;           // Maximum vote total

// Turn arbitrator window parameters

const int TUWIN_X = 570;            // x-coord of top
const int TUWIN_Y = 540;            // y-coord of left side
const int TUWIN_WIDTH = 175;        // Window width
const int TUWIN_HEIGHT = 50;        // Window height
const double TUWIN_MIN = -20.0;          // Minimum vote total
const double TUWIN_MAX = 20.0;           // Maximum vote total

// Power constants

const double CPU_FULL_VOLTAGE = 12.0;
const double CPU_DANGER_VOLTAGE = 11.0;

const double MOTOR_FULL_VOLTAGE = 12.0;
const double MOTOR_DANGER_VOLTAGE = 11.0;


/********** BEHAVIOR CONSTANTS **********/

// Bump halt

const int BUMP_SLEEP = 10;    // Number of seconds to sleep

// Recoil

const double RECOIL_SPEED = 100.0;
const double RECOIL_SPEED_SIGMA = 25.0;
const double RECOIL_TURN = 45.0;
const double RECOIL_TURN_SIGMA = 10.0;
const double RECOIL_WT = 10.0;

// Maintain alignment

```

```
const double MAX_BASE_TURRET_DEV = 1.0;

// Advance

const int ADV_SLOW_DIST = 60;
const int ADV_STOP_DIST = 6;
const int ADV_PER_SLOW_DIST = 12;
const int ADV_PER_STOP_DIST = 4;
const double ADV_SPEED = 75.0;
const double ADV_PER_SPEED = 20.0;
const double ADV_SPEED_SIGMA = 10.0;
const double ADV_SPEED_WT = 5.0;

// Advance slow

const int ADV_SLOW_STOP_DIST = 5;
const double ADV_SLOW_SPEED = 20.0;
const double ADV_SLOW_SPEED_SIGMA = 5.0;
const double ADV_SLOW_SPEED_WT = 5.0;

// Corridor advance

const int CORRIDOR_SPEED = 25;
const int CORRIDOR_SPEED_WIDE = 75;

// Maintain heading

const double MH_TURN_SIGMA = 45.0;
const double MH_TURN_WT = 1.0;

// Maintain transit heading

const double MTH_TURN_SIGMA = 45.0;
const double MTH_TURN_WT = 1.0;

// Avoid

const int AVOID_DIST = 36;
const double AVOID_TURN_SIGMA = 22.5;
const double AVOID_WT_FACTOR = 6.0;

// Transit avoid

const double TRANSIT_AVOID_TURN_SIGMA = 10.0;

// Avoid bias

const int AVOID_BIAS_DIST = 10;
const double AVOID_BIAS_ANGLE = 45.0;
const double AVOID_BIAS_SIGMA = 45.0;
const double AVOID_BIAS_WT = 1.0;

// Follow wall

const int FOLLOW_ABORT = 20;
const int FOLLOW_MAX_ALIGN_DIST = 40;
const int FOLLOW_STOP_DIST = 20;
const double FOLLOW_TURN_FACTOR = 0.2;
```

```
const double FOLLOW_TURN_SIGMA = 5.0;
const double FOLLOW_WT = 2.0;

// Maintain distance

const int DESIRED_DIST = 10;
const double MD_TURN_FACTOR = 0.2;
const double MD_TURN_SIGMA = 3.0;
const double MD_WT = 4.0;

// Follow path

const double NAV_MIN_ACT = 0.5;
const double NAV_SIGMA = 45.0;
const double NAV_WT = 5.0;

// Goal orient

const double GOAL_SIGMA = 45.0;
const double GOAL_WT = 5.0;

// Goal corridor orient

const double GOAL_CORRIDOR_NOISE = 5.0;          // Noise in turn angle

// Center home

const int CENTER_SPEED = 10;
const double CENTER_ERR_THRESH = 0.01;

// Angle localization

const int ANGLE_LOC_STEP = 10;
const int ANGLE_LOC_NUM_STEPS = 10;
const int ANGLE_LOC_SLEEP = 1;

/********** SEQUENCER CONSTANTS **********/

// 22.5 degrees between sonars

const int SONAR_SWEEP_WIDTH = 22;

// Sonar sweep speed

const int SONAR_SWEEP_SPEED = 20;

// Sonar sweep step (degrees)

const int SONAR_SWEEP_STEP = 2;

// Sonar sweep pause between steps (microseconds)

const unsigned int SONAR_SWEEP_PAUSE = 100000;

// Laser sweep step (degrees)

const int LASER_SWEEP_STEP = 5;
```

```
// Laser-limited sonar sweep speed

const int LLS_TURN_RATE = 150;

// Identification confirmation sequence (inches)

const double MAX_CONFIRM_DIST = 1.0;

// Local navigation sequencer tolerance (inches)

const double LOCAL_NAV_TOLERANCE = 18.0;

// Local navigation maximum timesteps for stall

const int STALL_TIMEOUT = 20;

// Angle above which local navigation turns robot in place

const double LOCAL_TIP_ANGLE = 90.0;

/********** CONTROL INTERFACE PARAMETERS **********/

const int NUM_CMD = 2;              // Number of command outputs

enum { SPEED, TURN };               // Command indexes

// Behavior modes

enum { EXPLORE_MODE, EXPLORE_LLS_MODE, NAVIGATION_MODE, TEST_MODE };

// Control commands

enum { CMD_EXPLORE,
       CMD_NAV, CMD_NAV_KBD, CMD_STOP, CMD_SAVE, CMD_LOAD,
       CMD_REDRAW, CMD_BUILD_GRID, CMD_SAVE_GRID, CMD_LOAD_GRID,
       CMD_GRID_IDENT, CMD_GRID, CMD_CLEAR, CMD_CLEAR_ROBOT,
       CMD_SONAR_SCAN, CMD_SONAR_SWEEP, CMD_SONAR_SWEEP_ABS,
CMD_CLEAR_SONAR,
       CMD_LASER_SCAN, CMD_LASER_SWEEP, CMD_LASER_SWEEP_ABS,
CMD_CLEAR_LASER,
       CMD_LLS_SCAN, CMD_LLS_SWEEP, CMD_LLS_SWEEP_ABS, CMD_CLEAR_LLS,
       CMD_GRID_UNDO, CMD_CENTER, CMD_PLACE_MAP,
       CMD_PLACE_IDENT, CMD_PLACE_GRID,
       CMD_LOCAL_NAV, CMD_ADD_PLACE, CMD_EDIT_PLACE,
       CMD_ADD_EDIT_LINK, CMD_DELETE_LINK,
       CMD_CLEAR_GLOBAL, CMD_SAVE_GLOBAL, CMD_LOAD_GLOBAL,
       CMD_DISPLAY_GLOBAL, CMD_GLOBAL_UNDO, CMD_INTEGRATE_GRID,
       CMD_FIND_FRONTIERS, CMD_DISPLAY_EDGES, CMD_DISPLAY_FRONTIERS,
       CMD_GOTO_FRONTIER, CMD_UPDATE_NAV_GRID, CMD_DETECT_CORRIDORS,
       CMD_CONNECT_CL, CMD_SEND_CL_GRID,
       CMD_BUMP,
       CMD_INIT_COMM, CMD_SEND_MSG, CMD_RECEIVE_MSG,
       CMD_EXIT };

/********** GRAPHICS CONSTANTS **********/

// Control window graphic parameters
```

```
const int CON_NUM_CMD = CMD_EXIT + 1;     // Number of command buttons

const int CON_WIN_LEFT = 850;       // x-coord of left side
const int CON_WIN_TOP = 0;          // x-coord of top

// Number of button columns
const int CON_COLS = 2;

// Number of button rows
const int CON_ROWS = (int) ((double) CON_NUM_CMD / 2.0 + 0.5);

const int CON_BUTTON_WIDTH = 150;   // Button width
const int CON_BUTTON_HEIGHT = 25;   // Button height

const int CON_LAB_WIDTH = 130;            // Label width
                              // (Must be less than button width)
const int CON_LAB_HEIGHT = 10;            // Label height
                              // (Must be less than button height)

// Evidence grid window screen boundaries

const int EGWIN_LEFT = 420;
const int EGWIN_TOP = 0;
const int EGWIN_RIGHT = 932;
const int EGWIN_BOTTOM = 512;

// Evidence grid window world coordinate boundaries

const int EGWIN_WC_LEFT = -3000;
const int EGWIN_WC_BOTTOM = -3000;
const int EGWIN_WC_RIGHT = 3000;
const int EGWIN_WC_TOP = 3000;

// Navigation grid window screen boundaries

const int NAV_WIN_LEFT = 420;
const int NAV_WIN_TOP = 0;
const int NAV_WIN_RIGHT = 932;
const int NAV_WIN_BOTTOM = 512;

// Navigation grid window world coordinate boundaries

const int NAV_WIN_WC_LEFT = -6000;
const int NAV_WIN_WC_BOTTOM = -6000;
const int NAV_WIN_WC_RIGHT = 6000;
const int NAV_WIN_WC_TOP = 6000;

/*
const int NAV_WIN_WC_LEFT = -3000;
const int NAV_WIN_WC_BOTTOM = -3000;
const int NAV_WIN_WC_RIGHT = 3000;
const int NAV_WIN_WC_TOP = 3000;
*/

// Global evidence grid window screen boundaries

const int GLOBAL_WIN_LEFT = 0;
const int GLOBAL_WIN_TOP = 0;
```

```
const int GLOBAL_WIN_RIGHT = 1024;
const int GLOBAL_WIN_BOTTOM = 1024;

/*
const int GLOBAL_WIN_LEFT = 0;
const int GLOBAL_WIN_TOP = 0;
const int GLOBAL_WIN_RIGHT = 512;
const int GLOBAL_WIN_BOTTOM = 512;
*/

// Evidence grid window world coordinate boundaries

const int GLOBAL_WIN_WC_LEFT = -6000;
const int GLOBAL_WIN_WC_BOTTOM = -6000;
const int GLOBAL_WIN_WC_RIGHT = 6000;
const int GLOBAL_WIN_WC_TOP = 6000;

/*
const int GLOBAL_WIN_WC_LEFT = -3000;
const int GLOBAL_WIN_WC_BOTTOM = -3000;
const int GLOBAL_WIN_WC_RIGHT = 3000;
const int GLOBAL_WIN_WC_TOP = 3000;
*/

// Color of robot in global window

static char GLOBAL_ROBOT_COLOR[STRLEN] = "Blue";

// Color of contact marker in global window

static char CONTACT_COLOR[STRLEN] = "Red";

// Size of contact marker in global window

const int CONTACT_MARK_SIZE = 50;

/********** FRONTIER CONSTANTS *********/

// Maximum number of frontiers

const int MAX_FRONTIERS = 1000;

// Radius of neighborhood around visited frontier (1/10 inch)

const double VISIT_RADIUS = 60.0;

// Radius of neighborhood around inaccesible frontier (1/10 inch)

const double INAC_RADIUS = 120.0;

// Maximum number of colors for blob coloring

const int MAX_COLORS = 1000;

// Number of colors to display

const int DISPLAY_COLORS = 16;

```

```
// Radius of region centroid marker

const double CENTROID_MARK_RADIUS = 75.0;

// Minimum region size

const int MIN_REGION_SIZE = 6;
// const int MIN_REGION_SIZE = 1;

// Maximum frontier distance

const double MAX_DIST = 100000.0;

// Frontier edge color

static char EDGE_COLOR[STRLEN] = "red";

// Color table

static char color_table[DISPLAY_COLORS][STRLEN] = {
  "Blue", "Green", "Gold", "Red",
  "SkyBlue", "LimeGreen", "Orange", "Magenta",
  "RoyalBlue", "Cyan", "LightCoral", "Violet",
  "SteelBlue", "Aquamarine", "Purple", "Turquoise" };

// Color conversions for robot window

static int robot_color[DISPLAY_COLORS] = {
  1, 6, 11, 16,
  2, 7, 12, 17,
  3, 8, 13, 18,
  4, 9, 15, 20 };

/********** NAVIGATION CONSTANTS **********/

// Distance to retreat on bumper contact (1/10 inch)

const int BUMP_RECOIL = 20;

// Speed of bump recoil

const int BUMP_RECOIL_SPEED = 20;

// Search status codes

const int SEARCH_SUCCESS = 0;
const int SEARCH_FAIL = 1;
const int SEARCH_TIMEOUT = 2;

// Maximum number of cells to search

const int SEARCH_MAX_CELLS = 10000;

// Maximum number of obstacle cells allowed in path region

const int MAX_OBS_COUNT = 2;

// Color of path in grid/global window
```

```
static char PATH_COLOR[ STRLEN] = "Red";
static char OPT_PATH_COLOR[ STRLEN] = "Blue";
static char TRAV_PATH_COLOR[ STRLEN] = "Red";

// Color of path in robot window

const int ROBOT_PATH_COLOR = 16;    // Red
const int ROBOT_OPT_PATH_COLOR = 2; // Blue
const int ROBOT_TRAV_PATH_COLOR = 16;     // Red

// Waypoint lookhead window (# waypoints)

const int WAYPOINT_WINDOW = 5;

// Number of waypoints between LLS sweeps during navigation

const int NAV_LLS_SWEEP_INTERVAL = 10000; // Never

/********** CORRIDOR CONSTANTS **********/

// Number of sensors to either side of sensor to check

const int CORRIDOR_SPAN = 3;

// Amount of forward space needed to be clear

const int CORRIDOR_FWD_RANGE = 12;

// Amount of space needed on sides of robot

const int CORRIDOR_SIDE_CLEARANCE = 6;

// Amount of forward space for wide corridor

const int CORRIDOR_WIDE_FWD_RANGE = 48;

// Amount of side space for a wide corridor

const int CORRIDOR_WIDE_SIDE_CLEARANCE = 24;

// Maximum deviation between corridor angle and goal bearing

const double CORRIDOR_MAX_DEVIATION = 90.0;

// Color of corridor in global window

static char CORRIDOR_COLOR[ STRLEN] = "Blue";
static char CORRIDOR_WIDE_COLOR[ STRLEN] = "Green";
static char CORRIDOR_SELECT_COLOR[ STRLEN] = "Red";
static char CORRIDOR_SELECT_WIDE_COLOR[ STRLEN] = "Gold";

// Color of corridor in robot window

const int CORRIDOR_COLOR_ROBOT = 2;             // Blue
const int CORRIDOR_WIDE_COLOR_ROBOT = 6;        // Green
const int CORRIDOR_SELECT_COLOR_ROBOT = 14;           // DeepPink
const int CORRIDOR_SELECT_WIDE_COLOR_ROBOT = 11;      // Yellow
```

```
/********** CONTINUOUS LOCALIZATION DECLARATIONS **********/

// Continuous localization host

static char CONTLOC_HOST[STRLEN] = "sun28";

// Continuous localization communication channel ID

const int CONTLOC_CHANNEL = 0;

// Minimum number of occupied cells for usable map

const int CONTLOC_MIN_OCC = 0;

// Exploration relocalization interval (inches)

const int EXPLORE_RELOC_DISTANCE = 96;

// Navigation relocalization interval (inches)

const int NAV_RELOC_DISTANCE = 24;

/********** MULTIROBOT DECLARATIONS **********/

// Message indicating new map exists

static char NEW_MAP_MSG[STRLEN] = "NEWMAP";

/********** MISCELLANEOUS DECLARATIONS **********/

// Status codes

const int OK = 0;
const int ALT = 1;
const int RETRY = 2;
const int ABORT = -1000;
const int TIMEOUT = -1001;
const int NO_PATH = -1002;
const int NO_FRONTIERS = -1003;

// External C functions

extern "C" int abs(int);
extern "C" int sleep(int);
extern "C" int usleep(unsigned int);
extern "C" void exit(int);
extern "C" void registergrids(Map3D map1, Map3D map2, double *dx,
                        double *dy, double *dt, double *fitness);

// Number of moving obstacles

const int NUM_MOB = 0;

// Length of experimental trial (steps)

//const int TRIAL_LENGTH = 10000;
```

```
const int TRIAL_LENGTH = 50000;

//const int TRIAL_LENGTH = 1000000000;    // Run forever

// Margin for random robot placement

const int RAND_MARGIN = 200;

class agent {

public:
  agent(void);                        // Constructor
  void control(void);                 // Main control loop

private:
  int bumped[NUM_TOUCH];              // BUMPER HACK ARRAY

  robot r;                     // Controlled robot
  arbitrator *speed_arb;              // Speed command arbitrator
  arbitrator *turn_arb;               // Turn command arbitrator
  place_net pnet;              // Place network
  char apndir[STRLEN];                // Name of APN directory

  mobstacle mob_list[NUM_MOB + 1];  // Moving obstacles

  Map3D global_grid;                  // Global evidence grid
  Map3D old_global;                   // Old global evidence grid
  Map3D egrid;                        // Evidence grid
  Map3D old_grid;              // Old evidence grid
  Map3D edge_grid;                    // Frontier edge grid
  Map3D nav_grid;              // Navigation grid

  int region_map[GLOBAL_X_RES][GLOBAL_Y_RES];    // Colored region grid

  int visit[NAV_X_RES][NAV_Y_RES];  // Visit map for path planning

  frontier frontiers[MAX_FRONTIERS];       // List of frontiers
  int num_front;               // Number of frontiers

  frontier front_visit[MAX_FRONTIERS];     // Visited frontiers
  int num_visit;               // Number of visited frontiers

  frontier front_inac[MAX_FRONTIERS];      // Inaccessible frontiers
  int num_inac;                       // Number of inaccessible frontiers

  int corridor[NUM_RANGE];            // Corridor detection array
  int wide_corridor[NUM_RANGE];              // Wide corridor detection
array

  CylSensorModelArray sonar_smd;     // Sonar sensor model
  CylSensorModelArray sonar_clear_smd;    // Sonar sensor model (clear)

  control_panel control_window;              // Control window
//  bar_graph speed_window;           // Speed command display
//  bar_graph turn_window;            // Turn command display
  window *grid_window;                // Evidence grid window (pointer)
//  window *nav_window;               // Navigation grid window (pointer)
  window *global_window;              // Global grid window (pointer)
```

```
  int global_refresh;                 // Set when global grid is displayed
  int realtime_display;               // Flags whether to display robot

  ofstream *logfile;                  // Log file

  int multi_mode;               // Multirobot mode (0:single, 1:multi)
  int contloc_mode;                   // Continuous localization mode
  int behavior_mode;                  // Behavior mode
  int home_dist;                // Path distance from home

  int destin;                         // Destination index

  int timer;                          // Total elapsed time (steps)

  double cpu_volt;                    // CPU battery voltage
  double motor_volt;                  // Motor battery voltage
  double cpu_min;               // Minimum CPU battery voltage
  double motor_min;                   // Minimum motor battery voltage

  double transit_dir;                 // Transit direction

  int pause_mode;

  int cell_count;               // Number of cells searched

  void reset(void);                   // Reset position and timer
  int user_command(void);             // Execute user command (if any)
  int iscan(void);                    // Scan for interrupt
  void terminate(void);               // End session
  void power_check(void);             // Check battery power

  // Behavior modes

  void bump_test(void);               // Bumper test
  void manual_exploration(void);      // Map territory under manual
control
  void exploration(void);             // Explore uncharted territory
  void exploration_lls(void);         // Explore uncharted territory (LLS)
  void reactive_exploration(void);    // Explore uncharted territory
  // reactively
  void navigation(void);              // Navigate to destination (mouse)
  void navigation_keyboard(void);     // Navigate to destination
(keyboard)
  void local_navigation(void);              // Navigate to local
destination point
  void frontier_navigate(void);             // Navigate to frontier
centroid

  // Explore uncharted territory (multiple trials)
  void multi_exploration(void);

  // Explore uncharted territory reactively (multiple trials)
  void multi_reactive_exploration(void);

  // Behavior sequencers

  // Manual exploration sequencer
```

```
void manual_exploration_seq(void);

// Exploration sequencer
void exploration_seq(void);

// Exploration sequencer (laser-limited sonar)
void exploration_lls_seq(void);

// Reactive exploration sequencer
void reactive_exploration_seq(void);

int navigation_seq(void);         // Navigation sequencer
void search_seq(void);            // Search sequencer
void map_seq(void);               // Build local grid
void center_seq(void);            // Move to center of current place

// Local navigation sequencer
int local_nav_seq(int x, int y);  // Local destination coordinates

// Local navigation sequencer for path following
int path_local_nav_seq(path p,          // Path to folloq
                 int &waypoint);  // Index of next waypoint

// Local navigation sequencer (continuous motion)
int local_cont_nav_seq(int x, int y);   // Local destination coords

// Local navigation sequencer (with alternate goal)
int local_nav_seq_alt(int gx, int gy,         // Goal coordinates
                int ax, int ay);  // Alternate goal coordinates

// Navigate to goal by planning and following path
int path_nav_seq(double gx, double gy); // World coords of goal

// Navigate to frontier by planning and following path
int frontier_path_nav_seq(int front_index);   // Frontier index

// Place identification sequencer
void ident_seq(void);

// Grid identification sequencer
void grid_ident_seq(void);

// Rotate and reset DR angle to match stored range image
void angle_loc_seq(int image[ NUM_RANGE] );

// Translate to match stored range image
void trans_loc_seq(int image[ NUM_RANGE] );

// Rotate sonar sensors and scan
void sonar_sweep_seq(Map3D map);

// Rotate sonar sensors and scan (absolute coordinates)
void sonar_sweep_abs_seq(Map3D map);

// Rotate laser scanner and scan
void laser_sweep_seq(Map3D map);

// Rotate laser scanner and scan (absolute coordinates)
```

```
  void laser_sweep_abs_seq(Map3D map);

  // Laser-limited sonar sweep
  void lls_sweep_seq(Map3D map);

  // Laser-limited sonar sweep (absolute coordinates)
  void lls_sweep_abs_seq(Map3D map);

  // Navigate to selected frontier
  int frontier_nav_seq(int front_index);  // Frontier destination index

  // Behavior sets

  // Behavior set for reactive exploration
  int reactive_explore_behaviors(void);

  int navigation_behaviors(void);         // Behavior set for navigation

  // Behavior set for local navigation
  int local_navigation_behaviors(int gx, int gy);

  // Behaviors

  /********** LOW-LEVEL BEHAVIORS **********/

  void bump_halt(void);       // Go limp if bumper touched
  void recoil(void);          // If touched in forward half, move
backward
  void bump_recoil(void);           // If bumper contact, recoil away
  void wander(void);          // Make small random turns
  int advance(void);          // Move forward unless front is blocked
  int advance_slow(void);     // Move forward slowly unless front is
blocked

  // Realign turret if it is not aligned with base
  void maintain_alignment(void);

  // Avoid nearby obstacles
  void avoid(void);

  // If front is completely blocked, bias avoidance toward the left side
  void avoid_bias_left(void);

  // If front is completely blocked, bias avoidance toward the right
side
  void avoid_bias_right(void);

  // Maintain current heading
  void maintain_heading(void);

  void veer(void);      // Veer away from potential collisions

  void follow_wall_right(void);     // Align with right wall
  void follow_wall_left(void);      // Align with left wall

  // Maintain desired distance from right wall
  void maintain_distance_right(void);
```

```
    // Maintain desired distance from left wall
    void maintain_distance_left(void);

    // Turn toward goal
    void goal_orient(int gx, int gy);

    /********** NAVIGATION BEHAVIORS **********/

    int follow_path(void);             // Turn to follow path
    int detect_dest(int destin);             // Detect arrival at
destination

    // Low-level commands

    void set_defaults(void);    // Set default command values
    void update(void);          // Update robot state and evidence grid
    void execute(void);         // Send commands to robot

    // Move obstacles
    void move_obstacles(void);

    // Delete obstacles
    void del_obstacles(void);

    // File access commands

    void save_net(void);               // Save network in file
    void load_net(void);               // Load network from file

    /********** LOCALIZATION FUNCTIONS **********/

    // Compute difference between image and range input
    double compute_range_err(int image[NUM_RANGE], vector rinput);

    // Compute translation vector between expected and actual position
    void trans_loc_vector(int image[NUM_RANGE], int &dx, int &dy);

    /********** EVIDENCE GRID FUNCTIONS **********/

    // Display evidence grid in X window
    void grid_display(window *win,     // Window pointer
                      Map3D map);      // Evidence grid

    // Display global evidence grid in X window
    void grid_display_global(Map3D map);    // Evidence grid

    // Display local grid for place
    void display_place_grid(void);

    // Display edge segments detected in evidence grid
    void grid_display_edges(int grid[GLOBAL_X_RES][GLOBAL_Y_RES]);

    // Display regions detected in evidence grid
    void grid_display_regions(int grid[GLOBAL_X_RES][GLOBAL_Y_RES]);

    // Display robot in window
    void display_robot(window *win,    // Window
                       int x, int y,   // Robot position (1/10 inch)
```

```
                 int theta,         // Robot heading (1/10 degree)
                 int turret); // Robot turret angle (1/10 degree)

  /********** FRONTIER FUNCTIONS **********/

  // Copy frontier <f2> to frontier <f1>
  void frontier_copy(frontier &f1, frontier f2);

  // Find frontiers in global grid
  void find_frontiers(void);

  // Find frontier edges in <raw> grid and store them in <edge> grid
  void find_frontier_edges(Map3D *raw,          // Raw evidence grid
(pointer)
                     Map3D *edge,         // Frontier edge grid
(pointer)
                     double height);      // Z-axis of edge plane

  // Find frontier regions in <edge> grid and add new frontiers
  void find_frontier_regions(Map3D edge,  // Frontier edge grid
                       double height);    // Z-coord of edge plane
        .

  // Segment <grid> image into regions in <color> using spreading
activation
  void spread_segment(Map3D grid,         // Uncolored grid
                  int color[GLOBAL_X_RES][GLOBAL_Y_RES], // Colored grid
                  double height);   // Z-coord of edge plane

  // Print colored grid cell values
  void print_region_map(int grid[GLOBAL_X_RES][GLOBAL_Y_RES]);    //
Colored grid

  // Determine size and centroid of frontier regions
  void analyze_regions(int grid[GLOBAL_X_RES][GLOBAL_Y_RES]);     //
Colored grid

  // Check whether centroid corresponds to previously visited frontier
  // Return 1 if visited, 0 otherwise
  int agent::visited(double cx, double cy);
                    // Centroid of new region

  // Check whether centroid corresponds to inaccessible frontier
  // Return 1 if inaccessible, 0 otherwise
  int agent::inaccessible(double cx, double cy);
                    // Centroid of new region

  // Return index of unvisited, accessible frontier closest to (x, y)
  // Return -1 if no such frontier exists
  int closest_frontier(double x, double y);

  // Mark region centroids in evidence grid window
  void display_region_centroids(double cx,      // Display center x-
coord
                        double cy); // Display center y-coord

  // Mark region centroids in robot window
  void display_robot_region_centroids(void);
```

```
   // Check whether cell (x, y) is part of frontier <front_index>
   int check_frontier_cell(int x, int y,   // Cell index
                     int front_index);     // Frontier index


   /******************** NAVIGATION FUNCTIONS ********************/

   // Move forward if front corridor is clear
   void corridor_advance(void);

   // Turn toward clear corridor closest to goal bearing
   void goal_corridor_orient(int gx, int gy);

   // Update navigation grid based on global grid
   void update_nav_grid(void);

   // Plan path to goal location (return 1 if path found, 0 otherwise)
   int path_plan(double wx, double wy,           // World coords of goal
            path &nav_path);        // Navigation path (optimized)

   // Plan path to goal location (return 1 if path found, 0 otherwise)
   int frontier_path_plan(double wx, double wy, // World coords of goal
                    int front_index,      // Frontier index
                    path &nav_path);       // Navigation path

   // Print all cells on path
   void print_path(path p);

   // Draw path in window
   void display_path(path p,         // Path
                 char *pcolor, // Path color
                 window *win); // Window

   // Draw path in robot window
   void display_path_robot(path p,         // Path
                     int pcolor);          // Path color

   // Find path from (sx, sy) to (gx, gy)
   int find_path(int sx, int sy,           // Start cell
            int gx, int gy,          // Goal cell
            path &p);          // Path

   // Find path from (sx, sy) to (gx, gy) or any point on frontier
<front_index>
   int frontier_find_path(int sx, int sy,  // Start cell
                    int gx, int gy,  // Goal cell
                    int front_index, // Frontier index
                    path &p);        // Path

   // Search cell (x,y) and return search status
   int search_cell(int x, int y,           // Search cell
             int gx, int gy, // Goal cell
             path &p);       // Path

   // Search cell (x,y) while navigating to frontier and return search
status
   int frontier_search_cell(int x, int y,        // Search cell
```

```
                     int gx, int gy,            // Goal cell
                     int front_index,           // Frontier index
                     path &p);                  // Path

  // Find index of (unvisited) neighbor closest to goal
  int closest_neighbor(int x, int y,            // Current cell index
                   int gx, int gy,        // Goal cell index
                   int &nx, int &ny);     // Next cell index

  // Reverse order of steps on path
  void reverse_path(path old_path,        // Initial path
                path &new_path);          // Reversed path


  // Optimize path by jumping between adjacent path cells
  void optimize_path(path old_path, // Initial path
                path &new_path);   // Optimized path

  // Convert path in grid cell coords to world coords
  void generate_world_path(path grid_path,      // Path in nav grid
                     path &world_path);   // Path in world coords

  // Initialize path
  void path_init(path &p);    // Path

  // Add point to path
  void path_add(path &p,      // Path
            int x, int y);    // Point to add to path

  //  Check to see whether region around point is free of known
obstacles
  int check_clear(int x, int y);

  //  Check to see whether region around point overlaps frontier
  int check_frontier_arrival(int x, int y, int front_index);

  // Finds waypoint furthest on path within destination tolerance, or
  // waypoint on path <p> closest to (x, y), returning the distance
  // to that point, and the waypoint's index in <index>
  double closest_waypoint(path p,         // Path
                    int x, int y,         // Current position (1/10
inch)
                    int index,            // Index of current waypoint
                    int &close_index);    // Index of closest waypoint

  /******************** CORRIDOR FUNCTIONS ********************/

  // Detect freespace cooridors in vicinity of robot
  void detect_corridors(void);

  // Check whether a corridor exists centered around sensor <center>
  // Return 1 if true, 0 otherwise
  int check_corridor(int center,    // Index of sensor in center of
corridor
                int fwd_range,     // Required forward space
                int side_clear);   // Required side space
```

```
// Check whether <sensor> is clear for cooridor <center>
int corridor_check_sensor(int center,         // Center sensor index
                          int sensor,         // Sensor index
                          int fwd_range,      // Required fwd space
                          int side_clear);    // Required side space

// Display corridors in robot window
void display_corridors(void);

// Display corridor boundaries centered around sensor <center>
void display_corridor(window *win,      // Window
                      int center, // Center sensor index
                      int fwd_range,    // Required forward space
                      int side_clear,   // Required side space
                      char *color);     // Corridor color

// Display corridor boundaries centered around sensor <center> in
robot window
void display_corridor_robot(int center, // Center sensor index
                            int fwd_range,  // Required forward space
                            int side_clear, // Required side space
                            int color); // Corridor color

// Select corridor nearest to specified heading
int select_corridor(double heading);    // Heading (degrees)

/********** INTERFACE TO CONTINUOUS LOCALIZATION **********/

// Initialize communications with continuous localization
void connect_cl(void);

// Send global grid to continuous localization
void send_cl_grid(void);

/********** MULTIROBOT COMMUNICATION **********/

// Initialize robot communication channel
void init_robot_comm(void);

// Send message to other robot
void send_robot_message(char *message);

// Send message to other robot (user mode)
void user_send_robot_message(void);

// BEGIN SCOUT THESIS CHANGE

// Receive message from other robot
// Returns 1 if message received, 0 otherwise
int receive_robot_message(int channel, char *message);

// END SCOUT THESIS CHANGE

// Receive message from other robot (user mode)
void user_receive_robot_message(void);

/********** MULTIROBOT EXPLORATION **********/
```

```
    // Integrate new map from remote robot (if a new map exists)
    void integrate_remote_map(void);
};

#endif
```

APPENDIX H. FRONTIER-BASED EXPLORATION CODE – AGENT.CC

This appendix contains the source code for the routine that controls the robot's exploration behaviors.

```
/*

  agent.cc

  Agent class
  Original code by Brian Yamauchi

  Modifications for SCOUT THESIS
  By Patrick A. Hillmeyer

*/

#include <iostream.h>
#include <math.h>
#include <string.h>

#include "agent.h"

// Arc direction strings

const char dir_str[16][20] =
{ "forward", "fwd-fwd-lf", "fwd-lf", "fwd-lf-lf",
    "left", "back-lf-lf", "back-lf", "back-back-lf",
    "back", "back-back-rt", "back-rt", "back-rt-rt",
    "right", "fwd-rt-rt", "fwd-rt", "fwd-fwd-rt" };

const char voice_str[16][STRLEN] =
{ "forward 0.", "forward 1.", "forward left 2.", "left 3.",
    "left 4.", "left 5.", "back left 6.", "back 7.",
    "back 8 .", "back 9.", "back right 10.", "right 11.",
    "right 12.", "right 13.", "forward right 14.", "forward 15." };

/********** AGENT CLASS CONSTRUCTOR **********/

agent::agent(void)
{

    char Global_Grid[STRLEN];         // Global Grid label
    char Control_Panel[STRLEN];             // Control Panel Label

    // Constructor

    char labels[MAX_CON][CON_LEN];
    int i;

    // Initialize mode flags

    multi_mode = 0;
    behavior_mode = EXPLORE_MODE;
    contloc_mode = 0;
    home_dist = 0;
    destin = 0;
```

```
    // Initialize graphics flags

    global_refresh = 1;
    realtime_display = 1;

    // Initialize frontier counters

    num_front = 0;
    num_inac = 0;

    // Initialize power variables

    cpu_volt = CPU_FULL_VOLTAGE;
    motor_volt = MOTOR_FULL_VOLTAGE;

    cpu_min = cpu_volt;
    motor_min = motor_volt;

    // Initialize abitrator windows

    speed_arb = new arbitrator(SPEED_RES, SPEED_MIN, SPEED_MAX,
SPEED_DEF, 0,
                         SPEED_NOISE);
    if (speed_arb == NULL) {
      cout << "agent::agent: Unable to allocate space for speed
arbitrator."
          << endl;
      exit(-1);
    }

    turn_arb = new arbitrator(TURN_RES, TURN_MIN, TURN_MAX, TURN_DEF, 1,
                        TURN_NOISE);
    if (turn_arb == NULL) {
      cout << "agent::agent: Unable to allocate space for turn
arbitrator."
          << endl;
      exit(-1);
    }

    //    speed_window.init(SPWIN_X, SPWIN_Y, SPWIN_WIDTH, SPWIN_HEIGHT,
"Speed",
    //                  SPEED_RES, SPWIN_MIN, SPWIN_MAX);
    //    turn_window.init(TUWIN_X, TUWIN_Y, TUWIN_WIDTH, TUWIN_HEIGHT,
"Turn",
    //                 TURN_RES, TUWIN_MIN, TUWIN_MAX);

    // Initialize control window

    strcpy(labels[CMD_EXPLORE], "EXPLORE");
    strcpy(labels[CMD_NAV], "NAVIGATE");
    strcpy(labels[CMD_NAV_KBD], "NAVIGATE (KBD)");
    strcpy(labels[CMD_STOP], "STOP");
    strcpy(labels[CMD_SAVE], "SAVE APN");
    strcpy(labels[CMD_LOAD], "LOAD APN");
    strcpy(labels[CMD_REDRAW], "DISPLAY APN");
    strcpy(labels[CMD_BUILD_GRID], "BUILD GRID");
    strcpy(labels[CMD_SAVE_GRID], "SAVE GRID");
```

```
    strcpy(labels[CMD_LOAD_GRID], "LOAD GRID");
    strcpy(labels[CMD_GRID], "DISPLAY GRID");
    strcpy(labels[CMD_CLEAR], "CLEAR GRID");
    strcpy(labels[CMD_CLEAR_ROBOT], "CLEAR ROBOT (ABS)");
    strcpy(labels[CMD_SONAR_SCAN], "SONAR SCAN");
    strcpy(labels[CMD_SONAR_SWEEP], "SONAR SWEEP");
    strcpy(labels[CMD_SONAR_SWEEP_ABS], "SONAR SWEEP (ABS)");
    strcpy(labels[CMD_CLEAR_SONAR], "CLEAR + SONAR SWEEP");
    strcpy(labels[CMD_LASER_SCAN], "LASER SCAN");
    strcpy(labels[CMD_LASER_SWEEP], "LASER SWEEP");
    strcpy(labels[CMD_LASER_SWEEP_ABS], "LASER SWEEP (ABS)");
    strcpy(labels[CMD_CLEAR_LASER], "CLEAR + LASER SWEEP");
    strcpy(labels[CMD_LLS_SCAN], "LLS SCAN");
    strcpy(labels[CMD_LLS_SWEEP], "LLS SWEEP");
    strcpy(labels[CMD_LLS_SWEEP_ABS], "LLS SWEEP (ABS)");
    strcpy(labels[CMD_CLEAR_LLS], "CLEAR + LLS SWEEP");
    strcpy(labels[CMD_GRID_UNDO], "UNDO SCAN/SWEEP");
    strcpy(labels[CMD_GRID_IDENT], "GRID IDENT");
    strcpy(labels[CMD_CENTER], "PLACE CENTER");
    strcpy(labels[CMD_PLACE_MAP], "PLACE MAP");
    strcpy(labels[CMD_PLACE_IDENT], "PLACE IDENT");
    strcpy(labels[CMD_PLACE_GRID], "DISPLAY PLACE GRID");
    strcpy(labels[CMD_LOCAL_NAV], "LOCAL NAVIGATION");
    strcpy(labels[CMD_ADD_PLACE], "ADD PLACE");
    strcpy(labels[CMD_EDIT_PLACE], "EDIT PLACE");
    strcpy(labels[CMD_ADD_EDIT_LINK], "ADD/EDIT LINK");
    strcpy(labels[CMD_DELETE_LINK], "DELETE LINK");
    strcpy(labels[CMD_CLEAR_GLOBAL], "CLEAR GLOBAL GRID");
    strcpy(labels[CMD_SAVE_GLOBAL], "SAVE GLOBAL GRID");
    strcpy(labels[CMD_LOAD_GLOBAL], "LOAD GLOBAL GRID");
    strcpy(labels[CMD_DISPLAY_GLOBAL], "DISPLAY GLOBAL GRID");
    strcpy(labels[CMD_GLOBAL_UNDO], "UNDO GLOBAL CHANGES");
    strcpy(labels[CMD_INTEGRATE_GRID], "INTEGRATE LOCAL GRID");
    strcpy(labels[CMD_FIND_FRONTIERS], "FIND FRONTIERS");
    strcpy(labels[CMD_DISPLAY_EDGES], "DISPLAY EDGES");
    strcpy(labels[CMD_DISPLAY_FRONTIERS], "DISPLAY FRONTIERS");
    strcpy(labels[CMD_GOTO_FRONTIER], "GO TO FRONTIER");
    strcpy(labels[CMD_UPDATE_NAV_GRID], "UPDATE NAV GRID");
    strcpy(labels[CMD_DETECT_CORRIDORS], "DETECT CORRIDORS");
    strcpy(labels[CMD_CONNECT_CL], "CONNECT TO CONTLOC");
    strcpy(labels[CMD_SEND_CL_GRID], "SEND CONTLOC GRID");
    strcpy(labels[CMD_BUMP], "BUMPER TEST");
    strcpy(labels[CMD_INIT_COMM], "INIT ROBOT COMM");
    strcpy(labels[CMD_SEND_MSG], "SEND MESSAGE");
    strcpy(labels[CMD_RECEIVE_MSG], "RECEIVE MESSAGE");

    strcpy(labels[CMD_EXIT], "EXIT");

//BEGIN SCOUT THESIS CHANGE
    sprintf(Control_Panel, "Control [%d] Panel", r.id);

    control_window.init_panel(CON_WIN_LEFT, CON_WIN_TOP,
CON_BUTTON_WIDTH,
                    CON_BUTTON_HEIGHT, Control_Panel,
                    CON_LAB_WIDTH, CON_LAB_HEIGHT, CON_NUM_CMD,
                    CON_COLS, CON_ROWS, labels);
    control_window.draw();

```

```
    // Initialize evidence grid window

    grid_window = new window(EGWIN_LEFT, EGWIN_TOP, EGWIN_RIGHT,
EGWIN_BOTTOM,
                           "Evidence Grid");
    grid_window->set_window(EGWIN_WC_LEFT, EGWIN_WC_BOTTOM,
EGWIN_WC_RIGHT,
                          EGWIN_WC_TOP);
    grid_window->iconify();

    // Initialize navigation grid window

    //    nav_window = new window(NAV_WIN_LEFT, NAV_WIN_TOP,
NAV_WIN_RIGHT,
    //                          NAV_WIN_BOTTOM, "Navigation Grid");
    //    nav_window->set_window(NAV_WIN_WC_LEFT, NAV_WIN_WC_BOTTOM,
NAV_WIN_WC_RIGHT,
    //                          NAV_WIN_WC_TOP);
    //    nav_window->iconify();

    // Initialize global evidence grid window

   sprintf(Global_Grid, "Global [%d] Grid", r.id);

    global_window = new window(GLOBAL_WIN_LEFT, GLOBAL_WIN_TOP,
                             GLOBAL_WIN_RIGHT, GLOBAL_WIN_BOTTOM,
                             Global_Grid);
// END SCOUT THESIS CHANGE
    global_window->set_window(GLOBAL_WIN_WC_LEFT, GLOBAL_WIN_WC_BOTTOM,
                            GLOBAL_WIN_WC_RIGHT, GLOBAL_WIN_WC_TOP);
    //    global_window->iconify();

    // Initialize evidence grid sensor models

//    cout << "Evidence grid: <disabled>" << endl;

    table_init();
    model_init(&sonar_smd, &sonar_clear_smd);

    // Initialize evidence grids

    grid_init(&egrid, 0.0, 0.0);
    grid_init(&old_grid, 0.0, 0.0);

    grid_init_nav(&nav_grid, 0.0, 0.0);

    grid_init_global(&global_grid, 0.0, 0.0);
    grid_init_global(&old_global, 0.0, 0.0);
    grid_init_global(&edge_grid, 0.0, 0.0);

    // Initialize moving obstacles

    for (i = 0; i < NUM_MOB; i++) {
      mob_list[i].rand_init();
    }

    // Reset timers

```

```
    timer = 0;

    // Initialize file pointers

    logfile = NULL;

    // Turn on all sensors

    r.sonar_on();
    r.ir_on();
    r.laser_on();

    // Initialize cell count

    cell_count = 0;

    // BUMPER FIX INITIALIZATION

    for (i = 0; i < NUM_TOUCH; i++) {
      bumped[ i] = 0;
    }
}

/********** USER CONTROL FUNCTIONS **********/

void agent::control(void)
{
    // Main control loop

  int quit = 0;

  do {
    quit = user_command();
  }
  while (!quit);
}


void agent::power_check(void)
{
  // Check battery power

  char vostr[ STRLEN] ;   // Voice string

  gs();

  cpu_volt = (double) (int) (voltCpuGet() * 100.0) / 100.0;
  motor_volt = (double) (int) (voltMotorGet() * 100.0) / 100.0;

  //  cout << "CPU voltage = " << cpu_volt << " : motor voltage = " <<
motor_volt
  //   << endl;

  //  cout << "CPU voltage = " << voltCpuGet() << " : motor voltage = "
  //       << voltMotorGet() << endl;

  if (cpu_volt < cpu_min) {
    cpu_min = cpu_volt;
```

```
      if (cpu_volt < CPU_DANGER_VOLTAGE) {
        sprintf(vostr, "Danger, Danger: C P U voltage is %.2f.\n",
cpu_volt);
        cout << vostr;
        tk(vostr);
      }
      else if (cpu_volt < CPU_FULL_VOLTAGE) {
        sprintf(vostr, "Warning: C P U voltage is %.2f.\n", cpu_volt);
        cout << vostr;
        tk(vostr);
      }
    }

    if (motor_volt < motor_min) {
      motor_min = motor_volt;
      if (motor_volt < MOTOR_DANGER_VOLTAGE) {
        sprintf(vostr, "Danger, Danger: Motor voltage is %.2f.\n",
motor_volt);
        cout << vostr;
        tk(vostr);
      }
      else if (motor_volt < MOTOR_FULL_VOLTAGE) {
        sprintf(vostr, "Warning: Motor voltage is %.2f.\n", motor_volt);
        cout << vostr;
        tk(vostr);
      }
    }
}

int agent::user_command(void)
{
  // Execute user command (if any)

  int quit = 0;          // Set to 1 for exit command
  int command;           // Command code

  //  power_check();

  control_window.refresh();
  command = control_window.scan_panel();

  switch(command) {
  case CMD_EXPLORE:
    exploration_lls();
    break;
  case CMD_NAV:
    navigation();
    break;
  case CMD_NAV_KBD:
    navigation_keyboard();
    break;
  case CMD_SAVE:
    save_net();
    break;
  case CMD_LOAD:
    load_net();
    break;
  case CMD_REDRAW:
```

```
      pnet.display();
      break;
    case CMD_BUILD_GRID:
      r.update();
      grid_clear(egrid);
      clear_robot(egrid, 0, 0);
      sonar_sweep_seq(egrid);
//      laser_sweep_seq(egrid);
      grid_display(grid_window, egrid);
      break;
    case CMD_SAVE_GRID:
      save_grid(egrid);
      break;
    case CMD_LOAD_GRID:
      load_grid(&egrid);
      r.update();
      grid_display(grid_window, egrid);
      break;
    case CMD_GRID:
      r.update();
      grid_display(grid_window, egrid);
      break;
    case CMD_CLEAR:
      grid_copy(old_grid, egrid);
      grid_clear(egrid);
      grid_display(grid_window, egrid);
      break;
    case CMD_CLEAR_ROBOT:
      grid_copy(old_grid, egrid);
      r.update();
      clear_robot(egrid, r.x, r.y);
      grid_display(grid_window, egrid);
      break;
    case CMD_SONAR_SCAN:
      grid_copy(old_grid, egrid);
      r.update();
//    SCOUT THESIS CHANGE - in line below changed r.turret to r.theta
      sonar_scan(sonar_smd, sonar_clear_smd, egrid, r.x, r.y, r.theta);
      grid_display(grid_window, egrid);
      break;
    case CMD_SONAR_SWEEP:
      grid_copy(old_grid, egrid);
      clear_robot(egrid, 0, 0);
      sonar_sweep_seq(egrid);
      grid_display(grid_window, egrid);
      break;
    case CMD_SONAR_SWEEP_ABS:
      grid_copy(old_grid, egrid);
      r.update();
      clear_robot(egrid, r.x, r.y);
      sonar_sweep_abs_seq(egrid);
      grid_display(grid_window, egrid);
      break;
    case CMD_CLEAR_SONAR:
      grid_copy(old_grid, egrid);
      grid_clear(egrid);
      r.update();
      clear_robot(egrid, 0, 0);
```

```
      sonar_sweep_seq(egrid);
      grid_display(grid_window, egrid);
      break;
    case CMD_LASER_SCAN:
      grid_copy(old_grid, egrid);
      r.update();
      //  Replaced r.turret with r.theta in line below
      laser_scan(egrid, r.x, r.y, r.theta);  // SCOUT THESIS change for
  Scout with fixed body laser
      grid_display(grid_window, egrid);
      break;
    case CMD_LASER_SWEEP:
      grid_copy(old_grid, egrid);
      r.update();
      laser_sweep_seq(egrid);
      grid_display(grid_window, egrid);
      break;
    case CMD_LASER_SWEEP_ABS:
      grid_copy(old_grid, egrid);
      r.update();
      laser_sweep_abs_seq(egrid);
      grid_display(grid_window, egrid);
      break;
    case CMD_CLEAR_LASER:
      grid_copy(old_grid, egrid);
      grid_clear(egrid);
      r.update();
      laser_sweep_seq(egrid);
      grid_display(grid_window, egrid);
      break;
    case CMD_LLS_SCAN:
      grid_copy(old_grid, egrid);
      r.update();
      lls_scan(sonar_smd, sonar_clear_smd, egrid, r.x, r.y, r.theta);  //
  SCOUT THESIS  - see change above
      grid_display(grid_window, egrid);
      break;
    case CMD_LLS_SWEEP:
      grid_copy(old_grid, egrid);
      r.update();
      clear_robot(egrid, 0, 0);
      lls_sweep_abs_seq(egrid);
      grid_display(grid_window, egrid);
      break;
    case CMD_LLS_SWEEP_ABS:
      grid_copy(old_global, global_grid);
      r.update();
      clear_robot(global_grid, 0, 0);
      lls_sweep_seq(global_grid);
      grid_display_global(global_grid);
      break;
    case CMD_CLEAR_LLS:
      grid_copy(old_grid, egrid);
      grid_clear(egrid);
      r.update();
      clear_robot(egrid, 0, 0);
      lls_sweep_seq(egrid);
      grid_display(grid_window, egrid);
```

```
        break;
      case CMD_GRID_UNDO:
        grid_copy(egrid, old_grid);
        grid_display(grid_window, egrid);
        break;
      case CMD_GRID_IDENT:
        grid_ident_seq();
        break;
      case CMD_CENTER:
        center_seq();
        break;
      case CMD_PLACE_MAP:
        map_seq();
        break;
      case CMD_PLACE_IDENT:
        ident_seq();
        break;
      case CMD_PLACE_GRID:
        display_place_grid();
        break;
      case CMD_LOCAL_NAV:
        local_navigation();
        break;
      case CMD_ADD_PLACE:
        pnet.add_place();
        break;
      case CMD_EDIT_PLACE:
        pnet.edit_place();
        break;
      case CMD_ADD_EDIT_LINK:
        pnet.add_edit_link();
        break;
      case CMD_DELETE_LINK:
        pnet.delete_link();
        break;
      case CMD_CLEAR_GLOBAL:
        grid_copy(old_global, global_grid);
        grid_clear(global_grid);
        grid_display_global(global_grid);
        num_front = 0;
        num_visit = 0;
        num_inac = 0;
        break;
      case CMD_SAVE_GLOBAL:
        save_grid(global_grid);
        break;
      case CMD_LOAD_GLOBAL:
        load_grid(&global_grid);
        r.update();
        grid_display_global(global_grid);
        break;
      case CMD_DISPLAY_GLOBAL:
        grid_display_global(global_grid);
        break;
      case CMD_GLOBAL_UNDO:
        grid_copy(global_grid, old_global);
        grid_display_global(global_grid);
        break;
```

```
    case CMD_INTEGRATE_GRID:
      integrate_grid(global_grid, egrid, (double) r.x / 120.0,
                     (double) r.y / 120.0, (double) r.theta / 10.0);
      grid_display_global(global_grid);
      break;
    case CMD_FIND_FRONTIERS:
      find_frontiers();
      break;
    case CMD_DISPLAY_EDGES:
      grid_display_global(global_grid);
      grid_display_edges(region_map);
      break;
    case CMD_DISPLAY_FRONTIERS:
      grid_display_global(global_grid);
      grid_display_regions(region_map);
      display_region_centroids(0.0, 0.0);
      //    display_robot_region_centroids();
      break;
    case CMD_GOTO_FRONTIER:
      frontier_navigate();
      break;
    case CMD_UPDATE_NAV_GRID:
      update_nav_grid();
      break;
    case CMD_DETECT_CORRIDORS:
      detect_corridors();
      display_corridors();
      break;
    case CMD_CONNECT_CL:
      connect_cl();
      break;
    case CMD_SEND_CL_GRID:
      send_cl_grid();
      break;
    case CMD_BUMP:
      bump_test();
      break;
    case CMD_INIT_COMM:
      init_robot_comm();
      break;
    case CMD_SEND_MSG:
      user_send_robot_message();
      break;
    case CMD_RECEIVE_MSG:
      user_receive_robot_message();
      break;
    case CMD_EXIT:
      terminate();
      quit = 1;
      break;
    }
    return(quit);
  }

  void agent::terminate(void)
  {
    // End session

```

```
    int i;

    // Delete mobstacles

    for (i = 0; i < NUM_MOB; i++) {
      mob_list[ i] .del_obs();
    }

    // Shut down robot

    r.shutdown();
}

int agent::iscan(void)
{
      // Scan for interrupt

      int command;

      control_window.refresh();
      command = control_window.scan_panel();
      if ((command == CMD_STOP) || (command == CMD_EXIT)) {
        st();
        return(ABORT);
      }
      else {
        return(OK);
      }
}

/********** BEHAVIOR CONTROL SYSTEMS **********/

void agent::bump_test(void)
{
    grid_display_global(global_grid);

    while(iscan() != ABORT) {
      update();
      bump_halt();
    }
}

void agent::manual_exploration(void)
{
    // Map territory under manual control

    int net_status; // Place net changed status

    timer = 0;
    behavior_mode = EXPLORE_MODE;

// BEGIN SCOUT THESIS CHANGE
    scout_vm(0, 0); // Necessary hack so robot will start moving later
// SCOUT THESIS CHANGE - changed pr to vm
// END SCOUT THESIS CHANGE

    manual_exploration_seq();
}
```

```
void agent::exploration(void)
{
  // Explore territory

  behavior_mode = EXPLORE_MODE;

  exploration_seq();
}

void agent::exploration_lls(void)
{
  // Explore territory using laser-limited sonar

  char comm_str[STRLEN];             // Contloc communication string

  behavior_mode = EXPLORE_LLS_MODE;

  // Set relocalization interval

  sprintf(comm_str, "reloc_distance = %d", EXPLORE_RELOC_DISTANCE);
  cout << "comm str = <" << comm_str << ">" << endl;
  write_comm(COMM_CHANNEL, comm_str, strlen(comm_str) + 1);

  // Exploration sequence

  exploration_lls_seq();
}

void agent::reactive_exploration(void)
{
  // Explore territory reactively

  int net_status; // Place net changed status
//  char logname[STRLEN];      // Log file name
//  char apnname[STRLEN];      // APN file name

  timer = 0;
  behavior_mode = EXPLORE_MODE;

/*  do {
    cout << "Enter log file name ==> ";
    cin >> logname;

    logfile = new ofstream(logname);
    if (logfile == NULL) {
      cout << "Unable to open log file <" << logname << ">." << endl;
    }
  }
  while(logfile == NULL);

  cout << "Enter APN file name ==> ";
  cin >> apnname;*/

//  reset();
//  pnet.clear_net();

  update();
```

```
  net_status = pnet.place_learn((double) r.x, (double) r.y,
                       (double) r.theta / 10.0);
  if (net_status & NEW_PLACE) {
    map_seq();
  }

  reactive_exploration_seq();

//  logfile->close();
//  pnet.save(apnname);
}

void agent::multi_exploration(void)
{
  // Explore territory (multiple trials)

  char prefix[ STRLEN] ;  // Filename prefixes
  char logname[ STRLEN] ; // Log filename
  char apnname[ STRLEN] ; // APN filename
  int trial_index;        // Trial index
  int trial_start;        // Index for initial trial
  int trial_end;  // Index for last trial
  int rand_x, rand_y, rand_heading; // Random initial position

  behavior_mode = EXPLORE_MODE;

  cout << "Enter filename prefix ==> ";
  cin >> prefix;

  cout << "Enter starting trial number ==> ";
  cin >> trial_start;

  cout << "Enter ending trial number ==> ";
  cin >> trial_end;

  for(trial_index = trial_start; trial_index <= trial_end;
trial_index++) {
    sprintf(logname, "%s%d.log", prefix, trial_index);
    sprintf(apnname, "%s%d.apn", prefix, trial_index);

    logfile = new ofstream(logname);
    if (logfile == NULL) {
      cout << "Unable to open log file <" << logname << ">." << endl;
    }
    else {
      cout << "Opening log file <" << logname << ">." << endl;
    }

    reset();
    pnet.clear_net();

    // Set random initial position

    rand_x = irand(PWIN_WC_LEFT + RAND_MARGIN, PWIN_WC_RIGHT -
RAND_MARGIN);
    rand_y = irand(PWIN_WC_BOTTOM + RAND_MARGIN, PWIN_WC_TOP -
RAND_MARGIN);
    rand_heading = irand(0, 3600);
```

```
    place_robot(rand_x, rand_y, rand_heading, rand_heading);

    // Hack to make sure robot isn't teleported into wall

// BEGIN SCOUT THESIS CHANGE
    scout_vm(1, 0);   // TEMP FIX- changed pr to vm
    scout_vm(-1, 0);   // TEMP FIX - changed pr to vm
// END SCOUT THESIS CHANGE

    update();
    pnet.place_learn((double) r.x, (double) r.y, (double) r.theta /
10.0);

    exploration_seq();

    if (logfile != NULL) {
      logfile->close();
    }
    pnet.save(apnname);
  }
}

void agent::multi_reactive_exploration(void)
{
  // Explore territory reactively (multiple trials)

  char prefix[STRLEN];  // Filename prefixes
  char logname[STRLEN]; // Log filename
  char apnname[STRLEN]; // APN filename
  int trial_index;      // Trial index
  int trial_start;      // Index for initial trial
  int trial_end;  // Index for last trial

  behavior_mode = EXPLORE_MODE;

  cout << "Enter filename prefix ==> ";
  cin >> prefix;

  cout << "Enter starting trial number ==> ";
  cin >> trial_start;

  cout << "Enter ending trial number ==> ";
  cin >> trial_end;

  for(trial_index = trial_start; trial_index <= trial_end;
trial_index++) {
    sprintf(logname, "%s%d.log", prefix, trial_index);
    sprintf(apnname, "%s%d.apn", prefix, trial_index);

    logfile = new ofstream(logname);
    if (logfile == NULL) {
      cout << "Unable to open log file <" << logname << ">." << endl;
    }
    else {
      cout << "Opening log file <" << logname << ">." << endl;
    }

    reset();
```

```
      pnet.clear_net();

      update();
      pnet.place_learn((double) r.x, (double) r.y, (double) r.theta /
  10.0);

      reactive_exploration_seq();

      if (logfile != NULL) {
        logfile->close();
      }
      pnet.save(apnname);
    }
  }

  void agent::navigation(void)
  {
    // Navigate to destination specified with mouse

    char comm_str[STRLEN];       // Contloc communication string
    char vostr[STRLEN];    // Voice string
    double gx, gy;  // Destination point (world coords)

    // Wait for user to click on destination in global window

    grid_display_global(global_grid);

    while(global_window->world_mouse(gx, gy) == 0);

    sprintf(vostr, "Navigating to %d, %d.\n", (int) gx, (int) gy);
    cout << vostr;
    tk(vostr);

    // Mark destination in window

    global_window->set_color("red");
    global_window->display_circle(gx, gy, CENTROID_MARK_RADIUS);
    global_window->display_line(gx - CENTROID_MARK_RADIUS, gy,
                                gx + CENTROID_MARK_RADIUS, gy);
    global_window->display_line(gx, gy - CENTROID_MARK_RADIUS,
                                gx, gy + CENTROID_MARK_RADIUS);
    global_window->set_color("black");

    // Set relocalization interval

    sprintf(comm_str, "reloc_distance = %d", NAV_RELOC_DISTANCE);
    cout << "comm str = <" << comm_str << ">" << endl;
    write_comm(COMM_CHANNEL, comm_str, strlen(comm_str) + 1);

    // Navigate to destination

    refresh_all();
    path_nav_seq(gx, gy);

    r.move_to_xy((int) gx, (int) gy);
    r.face_angle(0);

    tk("");   // Sometimes garbage gets stuck in the voice buffer
```

```
    sprintf(vostr, "Arrived at destination.\n");
    cout << vostr;
    tk(vostr);
  }

  void agent::navigation_keyboard(void)
  {
    // Navigate to destination specified with keyboard

    char comm_str[STRLEN];       // Contloc communication string
    char vostr[STRLEN];    // Voice string
    double gx, gy;  // Destination point (world coords)
    double gtheta;  // Destination orientation

    // Ask user to enter destination

    cout << "Enter destination (x, y, theta) (1/10 in, 1/10 deg) ==> ";
    cin >> gx >> gy >> gtheta;

    sprintf(vostr, "Navigating to %d, %d (%d).\n", (int) gx, (int) gy,
          (int) gtheta);
    cout << vostr;
    tk(vostr);

    // Mark destination in window

    grid_display_global(global_grid);

    global_window->set_color("red");
    global_window->display_circle(gx, gy, CENTROID_MARK_RADIUS);
    global_window->display_line(gx - CENTROID_MARK_RADIUS, gy,
                                gx + CENTROID_MARK_RADIUS, gy);
    global_window->display_line(gx, gy - CENTROID_MARK_RADIUS,
                                gx, gy + CENTROID_MARK_RADIUS);
    global_window->set_color("black");

    // Set relocalization interval

    sprintf(comm_str, "reloc_distance = %d", NAV_RELOC_DISTANCE);
    cout << "comm str = <" << comm_str << ">" << endl;
    write_comm(COMM_CHANNEL, comm_str, strlen(comm_str) + 1);

    // Navigate to destination

    refresh_all();
    path_nav_seq(gx, gy);

    r.move_to_xy((int) gx, (int) gy);
    r.face_angle((int) gtheta);

    tk("");   // Sometimes garbage gets stuck in the voice buffer

    sprintf(vostr, "Arrived at destination.\n");
    cout << vostr;
    tk(vostr);
  }
```

```
void agent::local_navigation(void)
{
    // Navigate to local coordinate point

    int x, y;                 // Local destination coordinates

    cout << "Enter destination point (x, y) ==> ";
    cin >> x >> y;

    local_nav_seq(x, y);
}

void agent::frontier_navigate(void)
{
  // Navigate to frontier centroid

  int front_index;      // Index of destination frontier

  if (num_front == 0) {
    cout << "No frontiers detected." << endl;
    return;
  }

  do {
    cout << "Enter frontier index ==> ";
    cin >> front_index;
    if ((front_index < 0) || (front_index >= num_front)) {
      cout << "Unknown frontier -- must be in range [ 0.." << num_front
<< "] ."
         << endl;
    }
  }
  while((front_index < 0) || (front_index >= num_front));

  frontier_nav_seq(front_index);
}

/********** BEHAVIORAL SEQUENCERS **********/

void agent::manual_exploration_seq(void)
{
  // Manual exploration sequencer

  int net_status; // Place net changed status

  cout << "Exploring under manual control..." << endl;

  do {
    update();
    net_status = pnet.place_learn((double) r.x, (double) r.y,
                      (double) r.theta / 10.0);

    if (net_status & NEW_PLACE) {
      cout << "Stop." << endl;
      tk("Stop.");
      st();
      ws(1, 1, 1, 5);
```

```
      map_seq();
    }
  }
  while(iscan() != ABORT);

  cout << "Exploration complete." << endl;
}

void agent::exploration_seq(void)
{
  // Exploration sequencer

  int front_index = 0;  // Frontier destination index
  int nav_status = OK;  // Navigation status

  cout << "Exploring..." << endl;
  tk("Exploring.");

  update();
  clear_robot(global_grid, r.x, r.y);
  sonar_sweep_abs_seq(global_grid);

  find_frontiers();

  while((num_front > 0) && (nav_status != ABORT) && (front_index != -1))
{
    front_index = closest_frontier((double) r.x, (double) r.y);

    if (front_index != -1) {
      nav_status = frontier_nav_seq(front_index);
      //    clear_robot(global_grid, r.x, r.y);
      //    sonar_sweep_seq(global_grid);
      find_frontiers();
    }
  }

  cout << "Exploration complete." << endl;
  tk("Exploration complete.");
}

void agent::exploration_lls_seq(void)
{
  // Exploration sequencer using laser-limited sonar


  char local_filename [STRLEN];              // Filename for local grid
  char local_posinfo [STRLEN];           // Position info for local grid
file
  char global_filename[STRLEN];      // Filename for global grid
  char global_posinfo[STRLEN];       // Position info for global grid
file
  char message[STRLEN];       // Message for multirobot communications

  double tx = 0.0, ty = 0.0;       // Registration translation vector
  double ttheta = 0.0;              // Registration rotation
  double score;                        // Registration score for local
grid

```

```
  int front_index = 0;  // Frontier destination index
  int nav_status = OK;  // Navigation status
  int occ;        // Number of occupied cells in global grid
  int unocc;            // Number of unoccupied cells in global grid

  cout << "Exploring..." << endl;
  tk("Exploring.");



// NEW SCOUT THESIS CHANGE below
// If robot is robot 1 it is the SERVER robot and will send its global
map out to
// the other CLIENT robots
// if robot is not number 1 then it is a CLIENT robot and will only
write its
// local scan to file
// in this way I hope to slow down error buildup in the map

 if (r.id == 1)  {

  sprintf(global_filename, "global%d.eg", r.id);

  // Sweep from initial position and find frontiers

  update();
  clear_robot(global_grid, r.x, r.y);

// BEGIN SCOUT THESIS CHANGE
// instead of using laser limited sonar use just the sonars

//  lls_sweep_abs_seq(global_grid);        //  commented out for Scout
  sonar_sweep_abs_seq(global_grid);   // use sonars only to explore
// END SCOUT THESIS CHANGE

  //  grid_display(grid_window, egrid);

  // Save global grid

  sprintf(global_posinfo, "%d %d %d", 0, 0, 0);
  save_grid_file(global_grid, global_filename, global_posinfo);

  // Notify other robot

  if (multi_mode) {
    send_robot_message(global_filename);
  }

  // Display global grid

  grid_display_global(global_grid);

  // Send grid to continuous localization

  grid_count_occ(global_grid, &occ, &unocc);
  cout << "Global grid cells: mapped = " << occ + unocc
       << " : occupied  = " << occ << endl;
  if (occ >= CONTLOC_MIN_OCC) {
```

```
    send_cl_grid();
  }

  // Check for new map from other robot

  if (multi_mode) {
    integrate_remote_map();
  }

  // Find initial frontiers

  find_frontiers();

  while(nav_status != ABORT) {
    if (num_front > 0) {
      // Navigate to closest frontier (index = -1 if inaccessible or
visited)

      front_index = closest_frontier((double) r.x, (double) r.y);
      if (front_index != -1) {
      nav_status = frontier_nav_seq(front_index);
      }
    }

    if ((num_front == 0) || (front_index == -1)) {
      if (iscan() == ABORT)  {        // add check for interrupts from
control panel
        nav_status = ABORT;
      }
      else  {
        cout << "No frontiers remaining, sweeping sensors..." << endl;
        tk("No frontiers, sweeping.");
        nav_status = NO_FRONTIERS;
       }
    }

    if ((nav_status != ABORT) && (nav_status != NO_PATH)) {
       clear_robot(global_grid, r.x, r.y);

// BEGIN SCOUT THESIS CHANGE
// instead of using laser limited sonar use just the sonars

//      lls_sweep_abs_seq(global_grid);   // commented out for Scout
       sonar_sweep_abs_seq(global_grid);
// END SCOUT THESIS CHANGE

      //    grid_display(grid_window, egrid);

      // Save global grid

      sprintf(global_posinfo, "%d %d %d", 0, 0, 0);
      save_grid_file(global_grid, global_filename, global_posinfo);

      // Notify other robot

      if (multi_mode) {
      send_robot_message(global_filename);
      }
```

```
        // Display global grid

        grid_display_global(global_grid);

        // Send grid to continuous localization

        grid_count_occ(global_grid, &occ, &unocc);

        cout << "Global grid cells: mapped = " << occ + unocc
           << " : occupied  = " << occ << endl;

        if (occ >= CONTLOC_MIN_OCC) {
        send_cl_grid();
        }

        // Check for new map from other robot

        if (multi_mode) {
        integrate_remote_map();
        }

        // Find new frontiers

        find_frontiers();
      }
    }

  }   // close for if r.id==1



// NEW MAJOR SCOUT THESIS change
// now handle the case of the CLIENT robots that just write their
// local maps

  else  {              // r.id != 1

   sprintf(local_filename, "local%d.eg", r.id);

   // Sweep from initial position and find frontiers

   update();
   grid_clear(egrid);     // clear the old local grid prior to scanning
   clear_robot(egrid, 0, 0);   // mark the cells under the robot as
unoccupied
//  clear_robot(global_grid, r.x, r.y);

// BEGIN SCOUT THESIS CHANGE
// instead of using laser limited sonar use just the sonars

//  lls_sweep_abs_seq(global_grid);         //  commented out for Scout
//  sonar_sweep_abs_seq(global_grid);   // use sonars only to explore in
global position
   sonar_sweep_seq(egrid);    // use sonars only to make local scan
centered around robot
// END SCOUT THESIS CHANGE
```

```
  //  grid_display(grid_window, egrid);

  // Register local grid with global grid - necessary when using robot
base position
  // for scanning vice global position

  tx = (double) r.x / 120.0;
  ty = (double) r.y / 120.0;
  ttheta = 0.0;

  // Save local grid

  sprintf(local_posinfo, "%d %d %d", r.x, r.y, 0);
  save_grid_file(egrid, local_filename, local_posinfo);

  // Notify other robot

  if (multi_mode) {
    send_robot_message(local_filename);
  }

//  Integrate local grid with global grid
  integrate_grid(global_grid, egrid, tx, ty, ttheta);


  // Display global grid
  grid_display_global(global_grid);

  // Send grid to continuous localization

  grid_count_occ(global_grid, &occ, &unocc);
  cout << "Global grid cells: mapped = " << occ + unocc
       << " : occupied  = " << occ << endl;
  if (occ >= CONTLOC_MIN_OCC) {
    send_cl_grid();
  }

  // Check for new map from other robot

  if (multi_mode) {
    integrate_remote_map();
  }

  // Find initial frontiers

  find_frontiers();

  while(nav_status != ABORT) {
    if (num_front > 0) {
      // Navigate to closest frontier (index = -1 if inaccessible or
visited)

      front_index = closest_frontier((double) r.x, (double) r.y);
      if (front_index != -1) {
      nav_status = frontier_nav_seq(front_index);
      }
    }

```

```
      if ((num_front == 0) || (front_index == -1)) {
        if (iscan() == ABORT)  {              // check for interrupts from
control panel
          nav_status = ABORT;
        }
        else  {
          cout << "No frontiers remaining, sweeping sensors..." << endl;
          tk("No frontiers, sweeping.");
          nav_status = NO_FRONTIERS;
        }
      }

      if ((nav_status != ABORT) && (nav_status != NO_PATH)) {
         grid_clear(egrid);
         clear_robot(egrid, 0, 0);

// BEGIN SCOUT THESIS CHANGE
// instead of using laser limited sonar use just the sonars

//       lls_sweep_abs_seq(global_grid);   // commented out for Scout
//        sonar_sweep_abs_seq(global_grid);
         sonar_sweep_seq(egrid);
// END SCOUT THESIS CHANGE

        //    grid_display(grid_window, egrid);

    // Register local grid with global grid - necessary when using robot
base position
    // for scanning vice global position

       tx = (double) r.x / 120.0;
       ty = (double) r.y / 120.0;
       ttheta = 0.0;



        // Save local grid

        sprintf(local_posinfo, "%d %d %d", r.x, r.y, 0);
        save_grid_file(egrid, local_filename, local_posinfo);

        // Notify other robot

        if (multi_mode) {
        send_robot_message(local_filename);
        }

    // Integrate local grid with global grid
        integrate_grid(global_grid, egrid, tx, ty, ttheta);

        // Display global grid

        grid_display_global(global_grid);

        // Send grid to continuous localization

        grid_count_occ(global_grid, &occ, &unocc);

```

```
        cout << "Global grid cells: mapped = " << occ + unocc
           << " : occupied  = " << occ << endl;

        if (occ >= CONTLOC_MIN_OCC) {
        send_cl_grid();
        }

        // Check for new map from other robot

        if (multi_mode) {
        integrate_remote_map();
        }

        // Find new frontiers

        find_frontiers();
      }
    }

  }  // close for else r.id != 1

//  END NEW MAJOR THESIS change



  cout << "Exploration complete." << endl;
  tk("Exploration complete.");
}



void agent::reactive_exploration_seq(void)
{
  // Reactive exploration sequencer

  cout << "Exploring reactively..." << endl;
  tk("Exploring");

  do {
    update();

    set_defaults();
    if (reactive_explore_behaviors() == 0) {
      execute();
    }
  }
  while((iscan() != ABORT) && (timer <= TRIAL_LENGTH));

  cout << "Exploration complete." << endl;
}

int agent::navigation_seq(void)
{
    // Follow path to destination
    // (returns ABORT if interrupt or error, OK otherwise)

    char vostr[ STRLEN] ;        // Voice string
    int suc[ PLACE_UNITS] ;      // Succesor list
```

```
    int gx, gy;                // Gateway location
    int arrived = 0;           // 1 when arrived at destination, 0
otherwise
    int nav_status = OK;       // Navigation status
    int i;

    behavior_mode = NAVIGATION_MODE;

    cout << "Navigating to place [" << destin << "]." << endl;
    sprintf(vostr, "Navigating to place %d.\n", destin);
    tk(vostr);

    //    ident_seq();

    //    cout << "Enter current place index ==> ";
    //    cin >> pnet.windex;

    while((pnet.windex != destin) && (nav_status != ABORT)) {
      pnet.find_paths(destin, suc);
      pnet.display();

      cout << endl << "Place transition list:" << endl;
      for (i = 0; i < pnet.num_units; i++) {
          cout << "[" << i << "] --> [" << suc[i] << "]" << endl;
      }
      cout << endl;

      if (suc[pnet.windex] == -1) {
          cout << "navigate_seq: No way to get from place [" <<
pnet.windex
            << "] to place [" << destin << "]." << endl;
          return(ABORT);
      }

      if (pnet.link[pnet.windex][suc[pnet.windex]] == NULL) {
          cout << "navigate_seq: Nonexistent link [" << pnet.windex
             << "] --> [" << suc[pnet.windex] << "]." << endl;
          return(ABORT);
      }

      gx = pnet.link[pnet.windex][suc[pnet.windex]]->gateway_x;
      gy = pnet.link[pnet.windex][suc[pnet.windex]]->gateway_y;

      cout << "Navigating to [" << pnet.windex << "] --> ["
           << suc[pnet.windex] << "] gateway at (" << gx << ", " << gy
<< ")."
           << endl;

      nav_status = local_nav_seq(gx, gy);

      if (nav_status != ABORT) {
        ident_seq();
      }
    }

    if (nav_status == ABORT) {
      cout << "Aborted." << endl;
    }
```

```
    else {
      cout << "Arrived at destination place [" << destin << "]." <<
endl;
    }
    return(nav_status);
}

int agent::local_nav_seq(int gx, int gy)     // Local destination
coordinates
{
  // Local navigation sequencer

  char vostr[STRLEN];           // Voice string
  double dist;                  // Distance from goal
  double min_dist;              // Minimum distance to goal so far
  double bearing;       // Bearing to goal
  int nav_status = 0;           // 1: arrived, 0: otherwise
  int stall_count = 0;          // Timesteps since progress made toward
goal

  update();

  dist = hypot((double) (gx - r.x), (double) (gy - r.y)) / 10.0;
  //  cout << "Distance from goal = " << dist << " inches" << endl;

  bearing = atan2((double) (gy - r.y), (double) (gx - r.x)) * RAD2DEG;
  //  cout << "Bearing to goal = " << bearing << endl;

  min_dist = dist;

  sprintf(vostr, "Navigating to %d %d.\n", gx, gy);
  //  tk(vostr);
  cout << vostr;

  if ((iscan() != ABORT) && (dist > LOCAL_NAV_TOLERANCE)) {
    r.face_angle_fast((int) (bearing * 10.0));
  }

  while((iscan() != ABORT) && (dist > LOCAL_NAV_TOLERANCE) &&
      (stall_count < STALL_TIMEOUT)) {

    update();

    bearing = atan2((double) (gy - r.y), (double) (gx - r.x)) * RAD2DEG;
    if (angle_diff(bearing, (double) r.theta / 10.0) > LOCAL_TIP_ANGLE)
{
      r.face_angle_fast((int) (bearing * 10.0));
    }
    else {
      set_defaults();
      nav_status = local_navigation_behaviors(gx, gy);
      execute();
    }

    dist = hypot((double) (gx - r.x), (double) (gy - r.y)) / 10.0;
    //    cout << "Distance from goal = " << dist << " inches" << endl;

    if (dist < min_dist) {
```

```
        min_dist = dist;
        stall_count = 0;
      }
      else {
        stall_count++;
        if (stall_count % 5 == 0) {
        sprintf(vostr, "Stalled for %d steps.\n", stall_count);
        cout << vostr;
        tk(vostr);
        }
      }
    }

    st();

    if (stall_count >= STALL_TIMEOUT) {
      sprintf(vostr, "Navigation timeout.\n",
            stall_count);
      cout << vostr;
      tk(vostr);
      return(TIMEOUT);
    }
    else if (dist > LOCAL_NAV_TOLERANCE) {
      cout << "Aborted." << endl;
      tk("Aborted.");
      return(ABORT);
    }

    cout << "Arrived." << endl;
    //  tk("Arrived.");
    return(OK);
  }

  int agent::path_local_nav_seq(path p,            // Path to folloq
                              int &waypoint)    // Index of next waypoint
  {
    // Local navigation sequencer for path following

    char message[STRLEN];        // Message from other robot
    char vostr[STRLEN];          // Voice string
    double dist;                 // Distance from goal
    double min_dist;             // Minimum distance to goal so far
    double close_dist;           // Distance to closest waypoint
    double bearing;        // Bearing to goal
    int gx, gy;                  // Waypoint coordinates
    int nav_status = 0;          // 1: arrived, 0: otherwise
    int stall_count = 0;         // Timesteps since progress made toward
  goal
    int close_index = waypoint; // Index of closest waypoint
    int i;

    // Set goal to next waypoint

    gx = p.x[waypoint];
    gy = p.y[waypoint];

    // Update robot state

```

```
  update();

  // Find distance/bearing to goal

  dist = hypot((double) (gx - r.x), (double) (gy - r.y)) / 10.0;
  //  cout << "Distance from goal = " << dist << " inches" << endl;

  bearing = atan2((double) (gy - r.y), (double) (gx - r.x)) * RAD2DEG;
  //  cout << "Bearing to goal = " << bearing << endl;

  min_dist = dist;

  sprintf(vostr, "Navigating to %d %d.\n", gx, gy);
  //  tk(vostr);
  cout << vostr;

  // Find distance to closest waypoint

  close_dist = closest_waypoint(p, r.x, r.y, waypoint, close_index);

  while((iscan() != ABORT) && (close_dist > LOCAL_NAV_TOLERANCE) &&
      (stall_count < STALL_TIMEOUT)) {

    // Update robot state

    update();

    // Stop if collision

    bump_halt();

    // Realign if turret is misaligned with base

    maintain_alignment();

    // Find bearing to goal

    bearing = atan2((double) (gy - r.y), (double) (gx - r.x)) * RAD2DEG;

    cout << "goal [" << waypoint << "] : bearing = " << bearing
       << " : dist = " << dist << " | closest [" << close_index
       << "] : dist = " << close_dist << endl;

    // Orient toward open corridor and advance

    goal_corridor_orient(gx, gy);
    corridor_advance();

    // Check distance from goal

    dist = hypot((double) (gx - r.x), (double) (gy - r.y)) / 10.0;

    if (dist < min_dist) {

      // If progress has been made, reset stall counter

      min_dist = dist;
      stall_count = 0;
```

```
      }
      else {

        // Otherwise, increment stall counter

        stall_count++;
        if (stall_count % 5 == 0) {
        sprintf(vostr, "Stalled for %d steps.\n", stall_count);
        cout << vostr;
        tk(vostr);
        }
      }

      // Find distance to closest waypoint

      close_dist = closest_waypoint(p, r.x, r.y, waypoint, close_index);
    }

    // Determine why navigation terminated

    if (stall_count >= STALL_TIMEOUT) {                  // Timeout
      sprintf(vostr, "Navigation timeout.\n",
            stall_count);
      cout << vostr;
      tk(vostr);
      return(TIMEOUT);
    }
    else if (close_dist > LOCAL_NAV_TOLERANCE) {         // User abort
      cout << "Aborted." << endl;
      tk("Aborted.");
      return(ABORT);
    }

    cout << "Arrived." << endl;                     // Success
    //  tk("Arrived.");

    // Advance to next waypoint on path after closest waypoint

    waypoint = close_index + 1;

    return(OK);
  }

  int agent::local_cont_nav_seq(int gx, int gy)     // Local destination
  coords
  {
    // Local navigation sequencer (continuous motion)

    char vostr[STRLEN];          // Voice string
    double dist;                 // Distance from goal
    double min_dist;             // Minimum distance to goal so far
    double bearing;        // Bearing to goal
    int nav_status = 0;          // 1: arrived, 0: otherwise
    int stall_count = 0;         // Timesteps since progress made toward
  goal

    update();
```

```
    dist = hypot((double) (gx - r.x), (double) (gy - r.y)) / 10.0;
    //  cout << "Distance from goal = " << dist << " inches" << endl;

    bearing = atan2((double) (gy - r.y), (double) (gx - r.x)) * RAD2DEG;
    //  cout << "Bearing to goal = " << bearing << endl;

    min_dist = dist;

    sprintf(vostr, "Navigating to %d %d.\n", gx, gy);
    //  tk(vostr);
    cout << vostr;

    if (angle_diff(bearing, (double) r.theta / 10.0) > LOCAL_TIP_ANGLE) {
      r.face_angle_fast((int) (bearing * 10.0));
    }

    while((iscan() != ABORT) && (dist > LOCAL_NAV_TOLERANCE) &&
        (stall_count < STALL_TIMEOUT)) {

      update();

      bearing = atan2((double) (gy - r.y), (double) (gx - r.x)) * RAD2DEG;
      if (angle_diff(bearing, (double) r.theta / 10.0) > LOCAL_TIP_ANGLE)
  {
        r.face_angle_fast((int) (bearing * 10.0));
      }
      else {
        set_defaults();
        nav_status = local_navigation_behaviors(gx, gy);
        execute();
      }

      dist = hypot((double) (gx - r.x), (double) (gy - r.y)) / 10.0;
      //    cout << "Distance from goal = " << dist << " inches" << endl;

      if (dist < min_dist) {
        min_dist = dist;
        stall_count = 0;
      }
      else {
        stall_count++;
        if (stall_count % 5 == 0) {
        sprintf(vostr, "Stalled for %d steps.\n", stall_count);
        cout << vostr;
        tk(vostr);
        }
      }
    }

    //  st();

    if (stall_count >= STALL_TIMEOUT) {
      sprintf(vostr, "Navigation timeout.\n",
            stall_count);
      cout << vostr;
      tk(vostr);
      return(TIMEOUT);
    }
```

```
  else if (dist > LOCAL_NAV_TOLERANCE) {
    cout << "Aborted." << endl;
    tk("Aborted.");
    return(ABORT);
  }

  cout << "Arrived." << endl;
  //  tk("Arrived.");
  return(OK);
}

int agent::local_nav_seq_alt(int gx, int gy,    // Goal coordinates
                       int ax, int ay)    // Alternate goal coordinates
{
  // Local navigation sequencer (with alternate goal)

  char vostr[ STRLEN] ;          // Voice string
  double dist;                   // Distance from goal
  double alt_dist;               // Distance from alternate goal
  double min_dist;               // Minimum distance to goal so far
  double bearing;       // Bearing to goal
  int nav_status = 0;            // 1: arrived, 0: otherwise
  int stall_count = 0;           // Timesteps since progress made toward
goal
  int interrupt;        // Interrupt code

  update();

  dist = hypot((double) (gx - r.x), (double) (gy - r.y)) / 10.0;
  //  cout << "Distance from goal = " << dist << " inches" << endl;

  alt_dist = hypot((double) (ax - r.x), (double) (ay - r.y)) / 10.0;
  //    cout << "Distance from alternate goal = " << alt_dist << "
inches"
  //   << endl;

  bearing = atan2((double) (gy - r.y), (double) (gx - r.x)) * RAD2DEG;
  //  cout << "Bearing to goal = " << bearing << endl;

  min_dist = dist;

  sprintf(vostr, "Navigating to %d %d.\n", gx, gy);
  //  tk(vostr);
  cout << vostr;

  cout << "Alternate goal: (" << ax << ", " << ay << ")" << endl;

  r.face_angle_fast((int) (bearing * 10.0));

  while(((interrupt = iscan()) != ABORT) && (dist > LOCAL_NAV_TOLERANCE)
&&
      (stall_count < STALL_TIMEOUT) && (alt_dist > LOCAL_NAV_TOLERANCE))
{

    update();

    bearing = atan2((double) (gy - r.y), (double) (gx - r.x)) * RAD2DEG;
```

```
      if (angle_diff(bearing, (double) r.theta / 10.0) > LOCAL_TIP_ANGLE)
{
        r.face_angle_fast((int) (bearing * 10.0));
      }
      else {
        set_defaults();
        nav_status = local_navigation_behaviors(gx, gy);
        execute();
      }

      dist = hypot((double) (gx - r.x), (double) (gy - r.y)) / 10.0;
      //    cout << "Distance from goal = " << dist << " inches" << endl;

      alt_dist = hypot((double) (ax - r.x), (double) (ay - r.y)) / 10.0;
      //    cout << "Distance from alternate goal = " << alt_dist << "
inches"
      //       << endl;

      if (dist < min_dist) {
        min_dist = dist;
        stall_count = 0;
      }
      else {
        stall_count++;
        if (stall_count % 5 == 0) {
        sprintf(vostr, "Stalled for %d steps.\n", stall_count);
        cout << vostr;
        tk(vostr);
        }
      }
    }

    st();

    if (stall_count >= STALL_TIMEOUT) {
      sprintf(vostr, "Navigation timeout.\n",
            stall_count);
      cout << vostr;
      tk(vostr);
      return(TIMEOUT);
    }

    if (interrupt == ABORT) {
      cout << "Aborted." << endl;
      tk("Aborted.");
      return(ABORT);
    }

    if (dist <= LOCAL_NAV_TOLERANCE) {
      cout << "Arrived." << endl;
      //    tk("Arrived.");
      return(OK);
    }

    if (alt_dist <= LOCAL_NAV_TOLERANCE) {
      cout << "Arrived at alternate goal." << endl;
      //    tk("Arrived at alternate goal.");
      return(ALT);
```

```
  }

  cout << "local_nav_seq_alt: Illegal termination." << endl;
  exit(-1);
}

void agent::center_seq(void)
{
    // Move to center of current place

    int cx, cy;          // Place center
    int ctheta;          // Place orientation

    if (pnet.windex == -1) {
      cout << "Unable to center at unknown location." << endl;
      return;
    }

    cx = (int) pnet.unit[ pnet.windex] .x;
    cy = (int) pnet.unit[ pnet.windex] .y;
    ctheta = (int) (pnet.unit[ pnet.windex] .theta * 10.0);

/*    cx = 0;
    cy = 0;
    ctheta = 0;*/

    r.move_to_xy(cx, cy);
    r.face_angle(ctheta);
// BEGIN SCOUT THESIS CHANGE
// comment out call for turret alignment - is not necessary for SCOUT
//    r.turret_align();
// END SCOUT THESIS CHANGE
}

int agent::path_nav_seq(double gx, double gy)   // World coords of goal
{
  // Navigate to goal by planning and following path

  path nav_path;                          // Navigation path
  int nav_status;                         // Navigation status
  int path_found;                         // 1 if path found, 0 otherwise
  int next_lls_point = NAV_LLS_SWEEP_INTERVAL;  // Waypoint for next LLS
sweep
  int i, j;

  path_found = path_plan(gx, gy, nav_path);
  if (!path_found) {
    return(NO_PATH);
  }

  //  cout << "Press <enter> to continue." << endl;
  //  cin.get();

  for (i = 1; i < nav_path.length; ) {
    nav_status = path_local_nav_seq(nav_path, i);

    // Stop immediately at end of path
    // (so the robot doesn't crash if the goal is next to a wall)
```

```
      if (i == nav_path.length) {
        st();
        cout << "Stopping at path's end." << endl;
      }

      if (i >= next_lls_point) {
        // Sweep laser at intervals (for contloc)
        lls_sweep_seq(egrid);

        next_lls_point += NAV_LLS_SWEEP_INTERVAL;
      }

      if (nav_status == ABORT) {             // User aborted
        return(ABORT);
      }

      if (nav_status == ALT) {         // Arrived unexpectedly at goal
        display_path(nav_path, TRAV_PATH_COLOR, global_window);
        return(OK);
      }

      if (nav_status == TIMEOUT) {     // Navigation timeout
        return(TIMEOUT);
      }

      // Mark traversed path segment in global window

      global_window->set_color(TRAV_PATH_COLOR);
      for (j = 0; j < i - 1; j++) {
        global_window->display_line(nav_path.x[j], nav_path.y[j],
                                    nav_path.x[j + 1], nav_path.y[j + 1]);
      }
      global_window->flush();
      global_window->set_color("black");

      // Mark traversed path segment in robot window

      //    for (j = 0; j < i - 1; j++) {
      //      draw_line(nav_path.x[j], nav_path.y[j],
      //                nav_path.x[j + 1], nav_path.y[j + 1],
      //                ROBOT_TRAV_PATH_COLOR + 2);
      //    }
    }

    st();

    return(OK);                            // Arrived at goal
  }

  int agent::frontier_path_nav_seq(int front_index)     // Frontier index
  {
    // Navigate to frontier by planning and following path

    path nav_path;  // Navigation path
    double gx, gy;  // World coords of frontier centroid
    int nav_status; // Navigation status
    int path_found; // 1 if path found, 0 otherwise
    int i, j;
```

```
    gx = frontiers[ front_index] .x;
    gy = frontiers[ front_index] .y;

    path_found = frontier_path_plan(gx, gy, front_index, nav_path);
    if (!path_found) {
      return(NO_PATH);
    }

    update();

    //  cout << "Press <enter> to continue." << endl;
    //  cin.get();

    for (i = 1; i < nav_path.length; ) {
      nav_status = path_local_nav_seq(nav_path, i);

      // Stop immediately at end of path
      // (so the robot doesn't crash if the goal is next to a wall)
      if (i == nav_path.length) {
        st();
        cout << "Stopping at path's end." << endl;
      }

      if (nav_status == ABORT) {              // User aborted
        return(ABORT);
      }

      if (nav_status == ALT) {          // Arrived unexpectedly at goal
        display_path(nav_path, TRAV_PATH_COLOR, global_window);
        return(OK);
      }

      if (nav_status == TIMEOUT) {      // Navigation timeout
        return(TIMEOUT);
      }

      // Mark traversed path segment in global window

      global_window->set_color(TRAV_PATH_COLOR);
      for (j = 0; j < i - 1; j++) {
        global_window->display_line(nav_path.x[ j] , nav_path.y[ j] ,
                                    nav_path.x[ j + 1] , nav_path.y[ j + 1] );
      }
      global_window->flush();
      global_window->set_color("black");

      // Mark traversed path segment in robot window

      //    for (j = 0; j < i - 1; j++) {
      //      draw_line(nav_path.x[ j] , nav_path.y[ j] ,
      //             nav_path.x[ j + 1] , nav_path.y[ j + 1] ,
      //             ROBOT_TRAV_PATH_COLOR + 2);
      //  }
    }

    return(OK);                         // Arrived at goal
  }
```

```
void agent::sonar_sweep_seq(Map3D map)
{
  // Rotate sonar sensors and scan

  int i;

  for (i = 0; i < SONAR_SWEEP_WIDTH; i += SONAR_SWEEP_STEP) {
    update();
// THESIS SCOUT CHANGE send r.theta not r.turret for SCOUT
    sonar_scan(sonar_smd, sonar_clear_smd, map, r.x, r.y, r.theta);
 //   cout << "r.theta= " << r.theta << endl;   // show robot heading
value
//    grid_display_global(map);  // TEMP FIX test map display - shows
updated display after each scan

// BEGIN SCOUT THESIS CHANGE
    scout_vm(0, SONAR_SWEEP_STEP * 10);   // Rotate the robot, - not the
turret  - changed pr to vm
//    ws(1, 1, 0, 5);         // TEMP FIX comment this line out and try
sleep instead
    sleep(3);   // SCOUT THESIS CHANGE added this line as test **PAUSE
robot at intervals**
  }

// SCOUT THESIS CHANGE - do not rotate Scout back as line below would do
//  hopefully this will decrease the odometry error buildup
//  scout_pr(0, SONAR_SWEEP_WIDTH * -10);     // Rotate the robot back
//  ws(1, 1, 0, 5);  // TEMP FIX comment this line out and try sleep
instead
//   sleep(3);   // TEMP FIX added this line as test
// END SCOUT THESIS CHANGE
  update();
}

void agent::sonar_sweep_abs_seq(Map3D map)
{
  // Rotate sonar sensors and scan (absolute coordinates)

  int i;

  for (i = 0; i < SONAR_SWEEP_WIDTH; i += SONAR_SWEEP_STEP) {
    update();
    sonar_scan_abs(sonar_smd, sonar_clear_smd, map, r.x, r.y, r.theta);
//    cout << "r.theta=" << r.theta << endl;   // show robot heading
value
//    grid_display_global(map);   // TEMP FIX test map display - shows
updated display after each scan
// TEMP FIX send r.theta not r.turret for SCOUT

// BEGIN SCOUT THESIS CHANGE
    scout_vm(0, SONAR_SWEEP_STEP * 10);      // changed pr to vm
//    ws(1, 1, 0, 5);      // TEMP FIX comment this line out and try
sleep cmd instead
    sleep(1);    //  TEMP FIX added this line as test
  }
```

```
//  SCOUT THESIS CHANGE - do not rotate Scout back as line below would
do
//  hopefully this will decrease the odometry error buildup
//  scout_pr(0, SONAR_SWEEP_WIDTH * -10);
//  ws(1, 1, 0, 5);     // TEMP FIX comment out this line and try sleep
cmd instead
//  sleep(3);   // TEMP FIX added this line as test
// END SCOUT THESIS CHANGE
   update();
}

void agent::laser_sweep_seq(Map3D map)
//  Rotate laser scanner and scan
{
   int scans = 0;

// BEGIN SCOUT THESIS CHANGE
   scout_vm(0, 3600);   // just in case we ever put a laser on the Scout
// TEMP FIX - use vm instead of pr
// END SCOUT THESIS CHANGE
   r.wait_start();

   while(State[ STATE_VEL_TURRET] > 0) {
     laser_scan(map, r.x, r.y, r.theta);   // TEMP FIX for SCOUT if it
ever has a fixed laser -yeh right
     if (realtime_display) {
       display_robot(global_window, State[ 34] , State[ 35] , State[ 36] ,
State[ 37] );
     }
     scans++;
   }

   cout << scans << " scans completed : avg scan interval = "
        << 360.0 / (double) scans << " degrees" << endl;
}

void agent::laser_sweep_abs_seq(Map3D map)
//  Rotate laser scanner and scan (absolute coordinates)
{
   int scans = 0;

// BEGIN SCOUT THESIS CHANGE
   scout_vm(0, 3600);   // TEMP FIX - use vm instead of pr
// END SCOUT THESIS CHANGE
   r.wait_start();

   while(State[ STATE_VEL_TURRET] > 0) {
     laser_scan_abs(map, r.x, r.y, r.theta);  //TEMP FIX for SCOUT with
fixed laser
     if (realtime_display) {
       display_robot(global_window, State[ 34] , State[ 35] , State[ 36] ,
State[ 37] );
     }
     scans++;
   }

   cout << scans << " scans completed : avg scan interval = "
        << 360.0 / (double) scans << " degrees" << endl;
```

```
}

void agent::lls_sweep_seq(Map3D map)
//  Laser-limited sonar sweep
{
  int scans = 0;

// SCOUT NOTE - do not know how Scout handles sp command
  sp(DEFAULT_SPEED, DEFAULT_TURN_RATE, 0);   // TEMP FIX for SCOUT

  r.sonar_single(0);
  r.ir_single(0);
// BEGIN SCOUT THESIS CHANGE
  scout_vm(0, 3600);      // TEMP FIX- try vm instead of pr commands for
SCOUT
// END SCOUT THESIS CHANGE
  r.wait_start();

  while(State[ STATE_VEL_TURRET] > 0) {
    lls_scan(sonar_smd, sonar_clear_smd, map, r.x, r.y, r.theta);   //
TEMP FIX for SCOUT
    if (realtime_display) {
      display_robot(global_window, State[ 34] , State[ 35] , State[ 36] ,
State[ 37] );
    }
    scans++;
  }

  cout << scans << " scans completed : avg scan interval = "
       << 360.0 / (double) scans << " degrees" << endl;

  r.ir_on();
  r.sonar_on();
  r.set_default_velocity();
}

void agent::lls_sweep_abs_seq(Map3D map)
//  Laser-limited sonar sweep (absolute coordinates)
{
  int scans = 0;

// SCOUT NOTE - do not know how Scout would handle sp command
  sp(DEFAULT_SPEED, DEFAULT_TURN_RATE, 0);   // TEMP FIX for SCOUT

  r.sonar_single(0);
  r.ir_single(0);
// BEGIN SCOUT THESIS CHANGE
  scout_vm(0, 3600);   // TEMP FIX - try vm instead of pr commands for
SCOUT
// END SCOUT THESIS CHANGE
  r.wait_start();

  while(State[ STATE_VEL_TURRET] > 0) {
    lls_scan_abs(sonar_smd, sonar_clear_smd, map, r.x, r.y, r.theta);
// TEMP FIX for SCOUT
    if (realtime_display) {
      display_robot(global_window, State[ 34] , State[ 35] , State[ 36] ,
State[ 37] );
```

```
      }
      scans++;
    }

    cout << scans << " scans completed : avg scan interval = "
         << 360.0 / (double) scans << " degrees" << endl;

    r.ir_on();
    r.sonar_on();
    r.set_default_velocity();
  }

  void agent::map_seq(void)
  {
    // Build local grid

    char vostr[ STRLEN] ;   // Voice string

    st();
  // BEGIN SCOUT THESIS CHANGE
    ws(1, 1, 0, 5);
  // END SCOUT THESIS CHANGE

    update();
    pnet.place_learn((double) r.x, (double) r.y, (double) r.theta / 10.0);

  //  sprintf(vostr, "Building map for place %d.\n", pnet.windex);
  //  cout << vostr;
  //  tk(vostr);

    grid_clear(pnet.unit[ pnet.windex] .lgrid);

    clear_robot(pnet.unit[ pnet.windex] .lgrid, 0, 0);
    sonar_sweep_seq(pnet.unit[ pnet.windex] .lgrid);
  //  laser_sweep_seq(pnet.unit[ pnet.windex] .lgrid);

    grid_display(grid_window, pnet.unit[ pnet.windex] .lgrid);

    cout << "Map complete." << endl;
  }

  void agent::ident_seq(void)
  {
      // Place identification sequencer

      // Build grid

      r.update();
      grid_clear(egrid);

      clear_robot(egrid, 0, 0);
      sonar_sweep_seq(egrid);
  //    laser_sweep_seq(egrid);

      grid_display(grid_window, egrid);

      // Identify grid

```

```
    grid_ident_seq();
}

void agent::grid_ident_seq(void)
{
  // Grid identification sequencer

  char comm_str[ STRLEN] ;              // Communications string
  char vostr[ STRLEN] ;           // Voice string
  double tx, ty;         // Translation vector
  double ttheta;         // Rotation
  int ix, iy, itheta;          // Identified position
  int ident;                   // Place ident index

  ident = pnet.best_match(egrid);

  cout << "Untransformed best match = [" << ident << "]" << endl;

  ident = pnet.best_trans_match(egrid, tx, ty, ttheta);
  cout << endl;

  cout << "Transformed best match = [" << ident << "]  (" <<
pnet.unit[ ident] .x
    << ", " << pnet.unit[ ident] .y << ")" << endl;
  cout << "Transformation = (" << tx << ", " << ty << ") [" << ttheta <<
"]"
    << endl;

  // Update dead reckoning

  gs();
  ix = (int) (tx * 120.0 + 0.5);
  iy = (int) (ty * 120.0 + 0.5);
  itheta = wrap(r.theta + (int) (ttheta * 10.0 + 0.5), 0, 3599);

  cout << endl;
  cout << "place = " << ident << " : x = " << ix << " : y = " << iy
       << " : theta = " << itheta << endl;

  sprintf(vostr, "I am at place %d.\n", ident);
  tk(vostr);

  pnet.windex = ident;
  pnet.display();

  r.x = ix + (int) (pnet.unit[ ident] .x + 0.5);
  r.y = iy + (int) (pnet.unit[ ident] .y + 0.5);
  r.theta = itheta;

  place_robot(r.x, r.y, r.theta, r.theta);

//  sprintf(comm_str, "%s/grid%d %d %d %d", apndir, ident, ix, iy,
itheta);
//  cout << "comm str = <" << comm_str << ">" << endl;
//  write_comm(COMM_CHANNEL, comm_str, strlen(comm_str) + 1);

//  while(read_comm(COMM_CHANNEL, comm_str, 80) < 1);
//  cout << "reply = < " << comm_str << ">" << endl;
```

```
}

int agent::frontier_nav_seq(int front_index)    // Frontier destination
index
{
  // Navigate to selected frontier

  int nav_status; // Navigation status
  char vostr[ STRLEN] ;   // Voice string

  cout << "Navigating to frontier [" << front_index << "] -- centroid ("
       << (int) frontiers[ front_index] .x << ", "
       << (int) frontiers[ front_index] .y << ")" << endl;

  sprintf(vostr, "Navigating to frontier %d.\n", front_index);
  tk(vostr);

  //  grid_display_global(global_grid);
  //  grid_display_regions(region_map);
  //  display_region_centroids(0.0, 0.0);
  //  display_robot_region_centroids();

  nav_status = frontier_path_nav_seq(front_index);

  if (nav_status == ABORT) {
    return(ABORT);
  }

  if (nav_status == OK) {
    sprintf(vostr, "Arrived at frontier %d.\n", front_index);
    cout << vostr;
    tk(vostr);

    if (num_visit == MAX_FRONTIERS) {
      cout << "Visited too many frontiers (>" << MAX_FRONTIERS << ")."
<< endl;
      exit(-1);
    }

    front_visit[ num_visit] .x = frontiers[ front_index] .x;
    front_visit[ num_visit] .y = frontiers[ front_index] .y;
    num_visit++;
  }

  if ((nav_status == TIMEOUT) || (nav_status == NO_PATH)) {
    if (num_inac > MAX_FRONTIERS) {
      cout << "frontier_nav_seq: Too many inaccessible frontiers (> " <<
      MAX_FRONTIERS << ")." << endl;
      exit(-1);
    }

    sleep(1);
    sprintf(vostr, "Frontier %d is inaccessible.\n", front_index);
    cout << vostr;
    tk(vostr);

    frontier_copy(front_inac[ num_inac] , frontiers[ front_index] );
    num_inac++;
```

```
  }

  return(nav_status);
}

/********** BEHAVIOR SETS **********/

int agent::reactive_explore_behaviors(void)
{
  // Behavior set for reactive exploration

  // Returns 1 if new place mapped, 0 otherwise

  int net_status = 0;

  advance();
  avoid();
  bump_halt();

  net_status = pnet.place_learn((double) r.x, (double) r.y,
                          (double) r.theta / 10.0);

  if (net_status & NEW_PLACE) {
    map_seq();
    return(1);
  }

  return(0);
}

int agent::navigation_behaviors(void)
{
    // Behavior set for navigation

  int nav_status; // 1 if active path link exists at current location,
                  // 0 otherwise

  advance();
  maintain_heading();
  avoid();
  bump_halt();

  nav_status = follow_path();

  pnet.place_recall((double) r.x, (double) r.y, (double) r.theta / 10.0,
                destin);

  return(nav_status);
}

int agent::local_navigation_behaviors(int gx, int gy)
{
  // Behavior set for local navigation

  int nav_status = 0;   // 1: arrived, 0: otherwise

  corridor_advance();
  return(nav_status);
```

```
}

/********** UTILITY FUNCTIONS **********/

void agent::reset(void)
{
  // Reset position and timer

  dp(0, 0);
  da(0, 0);

  timer = 0;
}

void agent::set_defaults(void)
{
    // Set default command values

    speed_arb->clear();
    turn_arb->clear();
}

void agent::update(void)
{
  // Update robot state and moving obstacles

  int i;

  if (timer % 10 == 0) {
    cout << "Time = " << timer << endl;
    //    power_check();
    //    if (logfile != NULL) {
    //      *logfile << timer << " " << pnet.num_units << endl;
    //    }
  }

  r.update();

  //  for (i = 0; i < NUM_MOB; i++) {
  //    mob_list[ i] .update(r.x, r.y);
  //  }

  //  clear_robot(global_grid, r.x, r.y);

  if (realtime_display) {
    display_robot(global_window, r.x, r.y, r.theta, r.theta);  // TEMP
FIX for SCOUT
  }

  //    sonar_print(egrid, 1);

  timer++;
}

void agent::execute(void)
{
    // Send commands to robot

```

```
    int speed_com, turn_com;

    //    speed_window.display(speed_arb->votes);
    //    turn_window.display(turn_arb->votes);

    speed_com = (int) speed_arb->command();
    turn_com = (int) (turn_arb->command() * 10.0);

    if ((speed_com == 0) && (turn_com == 0)) {
      turn_com = (int) (rdrand(-RAND_TURN, RAND_TURN) * 10.0);
//        cout << "Random turn <" << turn_com << ">" << endl;
    }

    r.move(speed_com, turn_com);

    home_dist += (int) speed_arb->command();
}

/********** BEHAVIORS **********/

void agent::bump_halt(void)
{
  // Go limp if bumper touched

// BEGIN SCOUT THESIS CHANGE
// comment out the code below that was a hack for a bad bumper
// rearrange code to match original code

  char vostr[STRLEN];   // Voice string
//  int touch_offset;   // Rotation offset for touch sensors
//  int abs_touch;      // Absolute index of tripped bumper
//  int sleepflag = 0;  // Do you sleep?
  int i;

  if (r.touch.max() > 0) {
    lp();   // robot motors stop

    for (i = 0; i < NUM_TOUCH; i++) {
      if (r.touch[i]) {
         sprintf(vostr, "Contact on bumper %d.", i);
         cout << vostr << endl;
         tk(vostr);
                        }   // close if
     }  // close for

     sprintf(vostr, "Sleeping for %d seconds.", BUMP_SLEEP);
     cout << vostr << endl;
     tk(vostr);

     sleep(BUMP_SLEEP);
  }  // close if
}  // clode bump_halt



// Below was all hack code for the procedure above

// HACK!  On Coyote, ignore multiple bumps on same bumper.
```

```
//
// REMOVE THIS WHEN COYOTE'S BUMPER BOARD IS FIXED
//
//    touch_offset = wrap((int) ((double) (r.theta + r.bumper_offset)
//                            / (double) BUMPER_SEP + 0.5),
//                        NUM_TOUCH);
//    abs_touch = wrap(i + touch_offset, NUM_TOUCH);
//
//    if ((r.id == 1) || !bumped[ abs_touch] ) {
//      lp();
//            sprintf(vostr, "Contact on bumper %d.", abs_touch);
//      cout << vostr << endl;
//      tk(vostr);
//      bumped[ abs_touch]  = 1;
//      sleepflag = 1;
//    }
//      }
//    }
//
//

//    if (sleepflag) {
//      sprintf(vostr, "Sleeping for %d seconds.", BUMP_SLEEP);
//      cout << vostr << endl;
//      tk(vostr);
//
//      sleep(BUMP_SLEEP);
//     }
//   }
// }

// END SCOUT THESIS CHANGE


// BEGIN SCOUT THESIS CHANGE
// NOTE - the following procedures recoil and bump_recoil were written
// for the Nomad 200 but are NOT implemented in the code
// The major problem with them on the Nomad 200 is misalignment
// of the turret and the base.
//  They should actually be easier to implement for the Nomad Scout
//  because of it lack of a turret bumpers will always be fixed in
relation
// to the robot.
//  Using these would be better than just using the bump_halt routine
// above which only stops the robot but does not get it away from the
obstacle

void agent::recoil(void)
{
    // If touched in forward half, move backward and turn

    double spd;
    double trn;

    if (r.touch.max(15, 5) > 0) {
      spd = rdrand(-RECOIL_SPEED, 0.0);
      trn = rdrand(-RECOIL_TURN, RECOIL_TURN);

      speed_arb->vote(spd, RECOIL_SPEED_SIGMA, RECOIL_WT);
```

```
      turn_arb->vote(trn, RECOIL_TURN_SIGMA, RECOIL_WT);

      cout << "Recoiling back... (speed = " << spd << ", turn = " << trn
          << ")" << endl;
    }
}

void agent::bump_recoil(void)
{
  // If bumper contact, recoil away

  char vostr[STRLEN];     // Voice string
  double rel_angle;       // Relative angle from robot to bumper contact
  double touch_angle;     // Absolute angle from robot to bumper contact
  double tx, ty;   // Coords of bumper contact
  int contact_flag = 0; // Contact indicator (0 = no, 1 = yes)
  int i;

  for (i = 0; i < NUM_TOUCH; i++) {
    if (r.touch[i]) {
      lp();        // Go limp

      sprintf(vostr, "Contact on bumper %d.", i);
      tk(vostr);

      // Compute contact angle

// NOTE - the BUMPER_SEP number here would have to be changed
//             to accomodate the different bumper pattern of the Scout
//              which is not evenly spaced around the robot
      rel_angle = (double) (i * BUMPER_SEP) / 10.0;

      sprintf(vostr, "Relative angle %.0f.", rel_angle);
      cout << vostr << endl;
      tk(vostr);

      if ((rel_angle <= 90.0) || rel_angle >= 270.0) {
      // Recoil back if contact is in forward half of robot

      cout << "<<< RECOILING BACK" << endl;
      tk("Recoiling back.");

// BEGIN SCOUT THESIS CHANGE
      scout_vm(-BUMP_RECOIL, 0);  // TEMP FIX - change pr to vm
//    ws(1, 1, 0, 10);  TEMP FIX - comment out the wait
// END SCOUT THESIS CHANGE
      }
      else {
      // Recoil forward if contact is in rear half of robot

      cout << "RECOILING FORWARD >>>" << endl;
      tk("Recoiling forward.");

// BEGIN SCOUT THESIS CHANGE
      scout_vm(BUMP_RECOIL, 0);  // TEMP FIX - change pr to vm
//    ws(1, 1, 0, 10);   TEMP FIX - comment out the wait
// END SCOUT THESIS CHANGE
      }
```

```
        contact_flag = 1;

        break;            // Only recoil from one contact
      }
    }

    // Update global grid for all contacts

    if (contact_flag) {
      for (i = 0; i < NUM_TOUCH; i++) {
        if (r.touch[ i] ) {
        // Compute contact position

// NOTE - the BUMPER_SEP number here would have to be changed
//              to accomodate the different bumper pattern of the Scout
//               which is not evenly spaced around the robot
        rel_angle = (double) (i * BUMPER_SEP) / 10.0;
        touch_angle = angle_wrap((double) r.theta / 10.0 + rel_angle);
        tx = (double) r.x + ROBOT_RADIUS * 120.0 * cos(touch_angle *
DEG2RAD);
        ty = (double) r.y + ROBOT_RADIUS * 120.0 * sin(touch_angle *
DEG2RAD);

        // Update global grid

        set_location(global_grid, tx / 120.0, ty / 120.0, SONAR_HEIGHT,
POS);
        }
      }

      grid_display_global(global_grid);
    }
}

void agent::maintain_alignment(void)
{
    // Realign turret if it is not aligned with base

    double dev;           // Deviation between base and turret angles
    int align_turn; // Turn required to realign turret

// BEGIN SCOUT THESIS CHANGE
// fake code into thinking nonexistent turret is aligned with base
   dev = 0.0;  // fix for SCOUT
//  dev = angle_diff((double) r.theta / 10.0, (double) r.turret / 10.0);

    if (dev > MAX_BASE_TURRET_DEV) {
      tk("Realigning.");
      st();

      do {
        cout << "REALIGNING:  base = " << r.theta << " : turret = "
           << r.turret << " : deviation = " << dev << endl;

        align_turn =
        (int) (angle_sgn_diff((double) r.theta / 10.0,
                             (double) r.turret / 10.0)
```

```
            * 10.0 + 0.5);
      cout << "Realignment turn = <" << align_turn << ">" << endl;


// NOTE - no turret on Scout to align, next two lines are ignored on
Scout
      scout_vm(0, 0);  // TEMP FIX for SCOUT
      ws(0, 0, 1, 100);

      update();
      dev = 0.0;   // fix for SCOUT
//      dev = angle_diff((double) r.turret / 10.0, (double) r.theta /
10.0);
    }
    while (dev > MAX_BASE_TURRET_DEV);

    cout << "Realignment complete: base = " << r.theta << " : turret = "
       << r.turret << " : deviation = " << dev << endl;
// END SCOUT THESIS CHANGE
  }
}

int agent::advance(void)
{
    // Move forward unless front is blocked (return 1 if blocked, 0
otherwise)

    int fwd_min;         // Minimum forward distance
    int per_min;         // Minimum peripheral distance
    double spd;                 // Desired speed

    fwd_min = r.arc[ FWD] ;
    per_min = r.range.min(FWD_RT, FWD_LF);

    if ((fwd_min <= ADV_STOP_DIST) || (per_min <= ADV_PER_STOP_DIST)) {
      speed_arb->vote(0.0, ADV_SPEED_SIGMA, ADV_SPEED_WT);
      return(1);
    }

    if ((fwd_min > ADV_SLOW_DIST) && (per_min > ADV_PER_SLOW_DIST)) {
      speed_arb->vote(ADV_SPEED, ADV_SPEED_SIGMA, ADV_SPEED_WT);
      return(0);
    }

    spd = ADV_SPEED;

    if (fwd_min <= ADV_SLOW_DIST) {
      spd = ADV_SPEED * (double) (fwd_min - ADV_STOP_DIST) /
      (double) (ADV_SLOW_DIST - ADV_STOP_DIST);
    }

    if ((per_min <= ADV_PER_SLOW_DIST) && (spd > ADV_PER_SPEED)) {
      spd = ADV_PER_SPEED;
    }

    speed_arb->vote(spd, ADV_SPEED_SIGMA, ADV_SPEED_WT);
    return(0);
}
```

```
int agent::advance_slow(void)
{
    // Move forward slowly unless front is blocked
    // (return 1 if blocked, 0 otherwise)

    int fwd_min;        // Minimum forward distance

    fwd_min = r.arc[ FWD] ;

    if (fwd_min > ADV_SLOW_STOP_DIST) {
      speed_arb->vote(ADV_SLOW_SPEED, ADV_SLOW_SPEED_SIGMA,
                ADV_SLOW_SPEED_WT);
      return(0);
    }
    else {
      speed_arb->vote(0.0, ADV_SLOW_SPEED_SIGMA, ADV_SLOW_SPEED_WT);
    }
}

void agent::maintain_heading(void)
{
    // Maintain current heading

    turn_arb->vote(0.0, MH_TURN_SIGMA, MH_TURN_WT);
}

void agent::avoid(void)
{
    // Avoid nearby obstacles

    int i;
    double wt;          // Voting weight for avoidance
    double theta; // Obstacle direction

    for (i = 0; i < NUM_RANGE; i++) {
      if (r.range[ i] < AVOID_DIST) {
          wt = AVOID_WT_FACTOR *
            (double) (AVOID_DIST - r.range[ i] ) / (double) AVOID_DIST;
          theta = r.sensor2theta(i);
          if (theta > 180.0) {
            theta -= 360.0;
          }
          turn_arb->vote(theta, AVOID_TURN_SIGMA, -wt);
      }
    }
}

void agent::avoid_bias_left(void)
{
    // If front is completely blocked, bias avoidance toward the left
side

    if (r.range.max(FWD_RT, FWD_LF) > AVOID_BIAS_DIST) {
      return;
    }

    turn_arb->vote(AVOID_BIAS_ANGLE, AVOID_BIAS_SIGMA, AVOID_BIAS_WT);
```

```
}

void agent::avoid_bias_right(void)
{
    // If front is completely blocked, bias avoidance toward the right
side

    if (r.range.max(FWD_RT, FWD_LF) > AVOID_BIAS_DIST) {
      return;
    }

    turn_arb->vote(-AVOID_BIAS_ANGLE, AVOID_BIAS_SIGMA, AVOID_BIAS_WT);
}

void agent::follow_wall_right(void)
{
    // Align with right wall

    double fturn; // Follow turn

    if ((r.range.min(BBR, FFR) > FOLLOW_MAX_ALIGN_DIST) ||
      (r.arc[ FWD] <= FOLLOW_STOP_DIST)) {
      return;
    }

//  cout << "min(RT,FWD) = <" << r.range.max(RT,FWD) << ">";

    if ((r.arc[ BACK_RT] != r.arc[ FWD_RT] ) && (r.arc[ FWD] >
FOLLOW_ABORT)) {
      fturn = FOLLOW_TURN_FACTOR * (double) (r.arc[ BACK_RT] -
r.arc[ FWD_RT] );
      turn_arb->vote(fturn, FOLLOW_TURN_SIGMA, FOLLOW_WT);
    }
//  cout << "" << endl;
}

void agent::follow_wall_left(void)
{
    // Align with right wall

    double fturn; // Follow turn

    if ((r.range.min(BBL, FFL) > FOLLOW_MAX_ALIGN_DIST) ||
      (r.arc[ FWD] <= FOLLOW_STOP_DIST)) {
      return;
    }

//  cout << "min(LF,FWD) = <" << r.range.max(LF,FWD) << ">";

    if ((r.arc[ BACK_LF] != r.arc[ FWD_LF] ) && (r.arc[ FWD] >
FOLLOW_ABORT)) {
      fturn= -FOLLOW_TURN_FACTOR * (double) (r.arc[ BACK_LF] -
r.arc[ FWD_LF] );
      turn_arb->vote(fturn, FOLLOW_TURN_SIGMA, FOLLOW_WT);
    }
//  cout << "" << endl;
}
```

```
void agent::maintain_distance_right(void)
{
    // Maintain desired distance from right wall

    int right_min;      // Minimum right range reading
    double mdturn;      // Maintain distance turn

    if ((r.range.min(BBR, FFR) > FOLLOW_MAX_ALIGN_DIST) ||
      (r.arc[ FWD] <= FOLLOW_STOP_DIST)) {
      return;
    }

    right_min = r.range.min(BACK, FWD);

    if (right_min != DESIRED_DIST) {
      mdturn = MD_TURN_FACTOR * (double) (DESIRED_DIST - right_min);
      turn_arb->vote(mdturn, MD_TURN_SIGMA, MD_WT);
//    cout << "right_min = <" << right_min << "> : turning <" <<
cmd[ TURN]
//       << ">" << endl;
    }
}

void agent::maintain_distance_left(void)
{
    // Maintain desired distance from left wall

    int left_min;
    double mdturn;      // Maintain distance turn

    if ((r.range.min(BBL, FFL) > FOLLOW_MAX_ALIGN_DIST) ||
      (r.arc[ FWD] <= FOLLOW_STOP_DIST)) {
      return;
    }

    left_min = r.range.min(FWD, BACK);

    if (left_min != DESIRED_DIST) {
      mdturn = -MD_TURN_FACTOR * (double) (DESIRED_DIST - left_min);
      turn_arb->vote(mdturn, MD_TURN_SIGMA, MD_WT);
//    cout << "left_min = <" << left_min << "> : turning <" << cmd[ TURN]
//       << ">" << endl;
    }
}

/********** NAVIGATION BEHAVIORS **********/

int agent::follow_path(void)
{
    // Turn to follow path

    // Returns 1 if outgoing place link is active, 0 otherwise

    double path_angle;         // Angle for navigation

    if (pnet.output_valid == 0) {
//    cout << "I'm lost..." << endl;
      return(0);
```

```
    }

    path_angle = angle_sgn_diff(pnet.output, (double) r.theta / 10.0);
    turn_arb->vote(path_angle, NAV_SIGMA, NAV_WT * pnet.conf);

    return(1);
}

int agent::detect_dest(int destin)
{
    // Detect arrival at destination

    if (pnet.windex == destin) {
      cout << "Arrived at destination." << endl;
      tk("Arrived at destination.");
      return(1);
    }
    else {
      return(0);
    }
}

void agent::goal_orient(int gx, int gy)
{
    // Turn toward goal (turn in place if deviation is too high)

    double bearing;            // Bearing from robot to goal
    double goal_angle;         // Angle between heading and bearing

    bearing = atan2((double) (gy - r.y), (double) (gx - r.x)) * RAD2DEG;

    //     cout << "goal = (" << gx << ", " << gy << ") : current = (" <<
r.x << ", "
    //        << r.y << ") : distance = "
    //        << hypot((double) (gy - r.y), (double) (gx - r.x)) / 10.0
    //          << " : bearing = " << bearing << endl;

    goal_angle = angle_sgn_diff(bearing, (double) r.theta / 10.0);
    turn_arb->vote(goal_angle, GOAL_SIGMA, GOAL_WT);

    //     cout << "heading = " << (double) r.theta / 10.0 << " :
goal_angle = "
    //       << goal_angle << endl;
}

/********** FILE ACCESS FUNCTIONS **********/

void agent::save_net(void)
{
  // Save net in directory

  char dirname[STRLEN];

  cout << "Enter directory name ==> ";
  cin >> dirname;

  pnet.save_all(dirname);
}
```

```
void agent::load_net(void)
{
    // Load net from directory

    cout << "Enter directory name ==> ";
    cin >> apndir;

    pnet.load_all(apndir);
    pnet.display();
}

/********** LOCALIZATION FUNCTIONS **********/

double agent::compute_range_err(int image[NUM_RANGE], vector rinput)
{
    // Compute difference between image and range input

    double match_err;
    int err_sum = 0;
    int i;

//    cout << "image/input:error = ";

    for (i = 0; i < NUM_RANGE; i++) {
      err_sum += abs(image[i] - rinput[i]);
//    cout << image[i] << "/" << rinput[i] << ":" <<
//        abs(image[i] - rinput[i]) << " ";
    }
//    cout << endl;

    match_err = (double) err_sum / (double) (NUM_RANGE * MAX_RANGE);
    cout << "match error = " << match_err << endl;

    return(match_err);
}

/******************** EVIDENCE GRID DISPLAY FUNCTIONS
********************/

void agent::grid_display(window *win,     // Window pointer
                         Map3D map) // Evidence grid
{
  // Display evidence grid in X window

  double xd, yd;               // Display coords
  double xscale, yscale, zscale;    // Cell dimensions (tenths of
inches)
  int x, y, z;                      // Cell index
  int xsize, ysize, zsize;          // Grid dimensions (# cells)
  int p;                       // Occupancy probability

  win->clear_window();

  xsize = map.msize[0];
  ysize = map.msize[1];
  zsize = map.msize[2];
```

```
  xscale = (map.himv[ 0]  - map.lomv[ 0] ) * 120.0 / (double) xsize;
  yscale = (map.himv[ 1]  - map.lomv[ 1] ) * 120.0 / (double) ysize;
  zscale = (map.himv[ 2]  - map.lomv[ 2] ) * 120.0 / (double) zsize;

//  cout << "Displaying grid (" << xsize << " x " << ysize << " x " <<
zsize
//    << ") : scale = (" << xscale << ", " << yscale << ", " << zscale
<< ")"
//      << endl;

  z = (int) ((SONAR_HEIGHT + HEIGHT_OFFSET - map.lomv[ 2] ) /
             (map.himv[ 2]  - map.lomv[ 2] ) * zsize);

  for (y = 0; y < ysize; y++) {
    for (x = 0; x < xsize; x++) {
      p = map.mapm[ z * xsize * ysize + y * xsize + x] ;

      xd = ((double) (x + 0.5) * xscale + map.lomv[ 0]  * 120.0);
      yd = ((double) (y + 0.5) * yscale + map.lomv[ 1]  * 120.0);

      if (p > 0) {
        win->display_circle(xd, yd, xscale / 4.0);
      }
      else if (p == 0) {
        win->display_point(xd, yd);
      }

      /*      if (p >= GRID_POS_THRESH) {
        win->display_circle(xd, yd, xscale / 4.0);
      }
      else if (p > GRID_NEG_THRESH) {
      if (p > 0) {
        win->set_color("blue");
        win->display_point(xd, yd);
        win->set_color("black");
      }
      else if (p < 0) {
        win->set_color("red");
        win->display_point(xd, yd);
        win->set_color("black");
      }
      else {
        win->display_point(xd, yd);
      }
      }*/

    }
  }

  win->draw_arc_buffer();
  win->flush();
}

void agent::grid_display_global(Map3D map)      // Evidence grid
{
  // Display global evidence grid in X window

  double xd, yd;              // Display coords
```

```
  double xscale, yscale, zscale;     // Cell dimensions (tenths of
inches)
  int x, y, z;                       // Cell index
  int xsize, ysize, zsize;           // Grid dimensions (# cells)
  int p;                       // Occupancy probability

  global_window->clear_window();

  xsize = map.msize[ 0] ;
  ysize = map.msize[ 1] ;
  zsize = map.msize[ 2] ;

  xscale = (map.himv[ 0]  - map.lomv[ 0] ) * 120.0 / (double) xsize;
  yscale = (map.himv[ 1]  - map.lomv[ 1] ) * 120.0 / (double) ysize;
  zscale = (map.himv[ 2]  - map.lomv[ 2] ) * 120.0 / (double) zsize;

  cout << "Displaying grid (" << xsize << " x " << ysize << " x " <<
zsize
    << ") : scale = (" << xscale << ", " << yscale << ", " << zscale <<
")"
      << endl;

  z = (int) ((SONAR_HEIGHT + HEIGHT_OFFSET - map.lomv[ 2] ) /
             (map.himv[ 2]  - map.lomv[ 2] ) * zsize);

  for (y = 0; y < ysize; y++) {
    for (x = 0; x < xsize; x++) {
      p = map.mapm[ z * xsize * ysize + y * xsize + x] ;

      xd = ((double) (x + 0.5) * xscale + map.lomv[ 0]  * 120.0);
      yd = ((double) (y + 0.5) * yscale + map.lomv[ 1]  * 120.0);

      if (p > 0) {
        global_window->display_circle(xd, yd, xscale / 4.0);
      }
      else if (p == 0) {
        global_window->display_point(xd, yd);
      }

      /*      if (p >= GRID_POS_THRESH) {
        global_window->display_circle(xd, yd, xscale / 4.0);
      }
      else if (p > GRID_NEG_THRESH) {
      if (p > 0) {
        global_window->set_color("blue");
        global_window->display_point(xd, yd);
        global_window->set_color("black");
      }
      else if (p < 0) {
        global_window->set_color("red");
        global_window->display_point(xd, yd);
        global_window->set_color("black");
      }
      else {
        global_window->display_point(xd, yd);
      }
      }*/

```

```
      }
    }

    global_window->draw_arc_buffer();
    global_window->flush();

    global_refresh = 1;
  }

  void agent::display_place_grid(void)
  {
    // Display local grid for place

    int index;            // Place index

    if (pnet.num_units == 0) {
      cout << "No places in APN." << endl;
      return;
    }

    cout << "Enter place index [0.." << pnet.num_units - 1 << "] ==> ";
    cin >> index;

    if ((index < 0) || (index >= pnet.num_units)) {
      cout << "Nonexistent place." << endl;
      return;
    }

    grid_display(grid_window, pnet.unit[index].lgrid);
  }

  void agent::grid_display_edges(int grid[GLOBAL_X_RES][GLOBAL_Y_RES])
                                 // Colored grid

  {
    // Display edge segments detected in evidence grid

    double xd, yd;              // Display coords
    double xscale, yscale;            // Cell dimensions (tenths of
inches)
    int x, y;                   // Cell index
    int xsize, ysize;                 // Grid dimensions (# cells)
    int p;                      // Occupancy probability

    xsize = GLOBAL_X_RES;
    ysize = GLOBAL_Y_RES;

    xscale = (GLOBAL_X_MAX - GLOBAL_X_MIN) * 120.0 / (double) xsize;
    yscale = (GLOBAL_Y_MAX - GLOBAL_Y_MIN) * 120.0 / (double) ysize;

//  cout << "Displaying grid (" << xsize << " x " << ysize << " x " <<
zsize
//    << ") : scale = (" << xscale << ", " << yscale << ", " << zscale
<< ")"
//      << endl;

    global_window->set_color(EDGE_COLOR);

```

```
    for (y = 0; y < ysize; y++) {
      for (x = 0; x < xsize; x++) {
        p = grid[ x][ y] ;

        xd = (double) (x + 0.5) * xscale + GLOBAL_X_MIN * 120.0;
        yd = (double) (y + 0.5) * yscale + GLOBAL_Y_MIN * 120.0;

        if (p > 0) {
        global_window->display_circle(xd, yd, xscale / 4.0);
        }
      }
    }

    global_window->set_color("black");

    global_window->draw_arc_buffer();
    global_window->flush();
}

void agent::grid_display_regions(int grid[ GLOBAL_X_RES][ GLOBAL_Y_RES] )
                                // Colored grid

{
    // Display regions detected in evidence grid

    double xd, yd;              // Display coords
    double xscale, yscale;            // Cell dimensions (tenths of
inches)
    int x, y;                   // Cell index
    int xsize, ysize;                 // Grid dimensions (# cells)
    int p;                      // Occupancy probability

    xsize = GLOBAL_X_RES;
    ysize = GLOBAL_Y_RES;

    xscale = (GLOBAL_X_MAX - GLOBAL_X_MIN) * 120.0 / (double) xsize;
    yscale = (GLOBAL_Y_MAX - GLOBAL_Y_MIN) * 120.0 / (double) ysize;

//  cout << "Displaying grid (" << xsize << " x " << ysize << " x " <<
zsize
//    << ") : scale = (" << xscale << ", " << yscale << ", " << zscale
<< ")"
//      << endl;

    for (x = 0; x < xsize; x++) {
      for (y = 0; y < ysize; y++) {
        p = grid[ x][ y] ;

        xd = (double) (x + 0.5) * xscale + GLOBAL_X_MIN * 120.0;
        yd = (double) (y + 0.5) * yscale + GLOBAL_Y_MIN * 120.0;

        if (p > 0) {
        global_window->set_color(color_table[ (grid[ x][ y]  - 1) %
                                     DISPLAY_COLORS] );
        //    cout << "display color = "
        //          << color_table[ (grid[ x][ y]  - 1) % DISPLAY_COLORS]  <<
endl;
        global_window->display_circle(xd, yd, xscale / 4.0);
```

```
        global_window->set_color("black");
      }
    }
  }

  global_window->draw_arc_buffer();
  global_window->flush();
}

void agent::display_robot(window *win,    // Window
                    int x, int y,   // Robot position (1/10 inch)
                    int theta,      // Robot heading (1/10 degree)
                    int turret)     // Robot turret angle (1/10 degree)
{
  // Display robot in window

  // Local constants

  const double robot_rad = ROBOT_RADIUS * 120.0; // Robot radius (1/10
inch)
  const double half_rad = robot_rad * 0.5;        // Half radius (1/10
inch)
  const double tendeg = 0.1 * DEG2RAD;            // 1/10 degree in
radians

  static double old_fx, old_fy;            // Old robot position
  static double old_ftheta;         // Old robot heading
  static double old_fturret;        // Old robot turret angle
  static double old_bx, old_by;            // Old endpoint of base line
  static double old_tx1, old_ty1,   // Old endpoints of turret line
    old_tx2, old_ty2;

  double fx, fy;        // Robot position (floating point)
  double ftheta;        // Robot heading (floating point)
  double fturret;       // Robot turret angle (floating point)
  double bx, by;        // Endpoint of base line
  double tx1, ty1, tx2, ty2;  // Endpoints of turret line

  fx = (double) x;
  fy = (double) y;
  ftheta = (double) theta;
  fturret = (double) turret;

  bx = fx + cos(ftheta * tendeg) * half_rad;
  by = fy + sin(ftheta * tendeg) * half_rad;

  tx1 = fx + cos(fturret * tendeg) * half_rad;
  ty1 = fy + sin(fturret * tendeg) * half_rad;

  tx2 = fx + cos(fturret * tendeg) * robot_rad;
  ty2 = fy + sin(fturret * tendeg) * robot_rad;

  if (!global_refresh) {
    win->display_xor_circle(old_fx, old_fy, robot_rad);
    win->display_xor_line(old_fx, old_fy, old_bx, old_by);
    win->display_xor_line(old_tx1, old_ty1, old_tx2, old_ty2);
  }
  global_refresh = 0;
```

```
    win->display_xor_circle(fx, fy, robot_rad);
    win->display_xor_line(fx, fy, bx, by);
    win->display_xor_line(tx1, ty1, tx2, ty2);

    win->flush();

    old_fx = fx;
    old_fy = fy;
    old_ftheta = ftheta;
    old_fturret = fturret;

    old_bx = bx;
    old_by = by;

    old_tx1 = tx1;
    old_ty1 = ty1;
    old_tx2 = tx2;
    old_ty2 = ty2;
  }

  /******************** FRONTIER FUNCTIONS ********************/

  void agent::frontier_copy(frontier &f1, frontier f2)
  {
    // Copy frontier <f2> to frontier <f1>

    f1.x = f2.x;
    f1.y = f2.y;
    f1.size = f2.size;
    f1.color = f2.color;
  }

  void agent::find_frontiers(void)
  {
    // Find frontiers in global grid

    find_frontier_edges(&global_grid, &edge_grid, SONAR_HEIGHT);
    find_frontier_regions(edge_grid, SONAR_HEIGHT);
    //  grid_display_global(global_grid);
    grid_display_regions(region_map);
    display_region_centroids(0.0, 0.0);
    //  display_robot_region_centroids();
  }

  void agent::find_frontier_edges(Map3D *raw,      // Raw evidence grid
  (pointer)
                              Map3D *edge,      // Frontier edge grid
  (pointer)
                              double height)    // Z-coord of edge plane
  {
    // Find frontier edges in <raw> grid and store them in <edge> grid

    int xsize, ysize, zsize;    // Grid dimensions (# cells)
    int x, y, z;                // Cell index
    int p;                  // Occupancy probability
    int unk;                // Unknown neighbor flag (0 = true)

```

```
  xsize = raw->msize[ 0] ;
  ysize = raw->msize[ 1] ;
  zsize = raw->msize[ 2] ;

  if ((xsize != edge->msize[ 0] ) || (ysize != edge->msize[ 1] ) ||
      (zsize != edge->msize[ 2] )) {
    cout << "find_frontier_edges: Grid size mismatch." << endl;
    return;
  }

  z = (int) ((height + HEIGHT_OFFSET - raw->lomv[ 2] ) /
          (raw->himv[ 2]  - raw->lomv[ 2] ) * zsize);

  for (x = 1; x < xsize - 1; x++) {
    for (y = 1; y < ysize - 1; y++) {
      edge->mapm[ z * xsize * ysize + y * xsize + x] = 0;

      p = raw->mapm[ z * xsize * ysize + y * xsize + x] ;

      if (p < 0) {

      // unk = 0 if and only if one of cell (x,y)'s neighbors is unknown

      unk = raw->mapm[ z * xsize * ysize + y * xsize + x - 1] *
        raw->mapm[ z * xsize * ysize + y * xsize + x + 1] *
        raw->mapm[ z * xsize * ysize + (y - 1) * xsize + x] *
        raw->mapm[ z * xsize * ysize + (y + 1) * xsize + x] ;

      if (unk == 0) {
        edge->mapm[ z * xsize * ysize + y * xsize + x] = 1;
      }
      }

      /*       if (p <= GRID_NEG_THRESH) {
      if (((raw->mapm[ z * xsize * ysize + y * xsize + x - 1]
            > GRID_NEG_THRESH) &&
           (raw->mapm[ z * xsize * ysize + y * xsize + x - 1]
            < GRID_POS_THRESH)) ||
          ((raw->mapm[ z * xsize * ysize + y * xsize + x + 1]
            > GRID_NEG_THRESH) &&
           (raw->mapm[ z * xsize * ysize + y * xsize + x + 1]
            < GRID_POS_THRESH)) ||
          ((raw->mapm[ z * xsize * ysize + (y - 1) * xsize + x]
            > GRID_NEG_THRESH) &&
           (raw->mapm[ z * xsize * ysize + (y - 1) * xsize + x]
            < GRID_POS_THRESH)) ||
          ((raw->mapm[ z * xsize * ysize + (y + 1) * xsize + x]
            > GRID_NEG_THRESH) &&
           (raw->mapm[ z * xsize * ysize + (y + 1) * xsize + x]
            < GRID_POS_THRESH))) {
        edge->mapm[ z * xsize * ysize + y * xsize + x] = 1;
      }
      }*/

    }
  }
}

```

```
void agent::find_frontier_regions(Map3D edge,         // Frontier edge
grid
                          double height)  // Z-coord of edge plane
{
  // Find frontier regions in <edge> grid and add new frontiers

  spread_segment(edge, region_map, height);
  analyze_regions(region_map);
}

void agent::spread_segment(Map3D grid,          // Uncolored grid
                     int color[GLOBAL_X_RES][GLOBAL_Y_RES],
                        // Colored grid
                     double height) // Z-coord of edge plane
{
  // Segment <grid> image into regions in <color> using spreading
activation

  int x, y, z;                 // Cell index
  int nx, ny;                  // Neighboring cell index
  int num_colors = 1;          // Number of colors
  int xsize, ysize, zsize;     // Grid dimensions (# cells)
  int changed;                 // Flag indicating whether cell colors
changed

  // Find grid dimensions

  xsize = grid.msize[0];
  ysize = grid.msize[1];
  zsize = grid.msize[2];

  z = (int) ((height + HEIGHT_OFFSET - grid.lomv[2]) /
         (grid.himv[2] - grid.lomv[2]) * zsize);

  // Set initial colors

  for (x = 0; x < xsize; x++) {
    for (y = 0; y < ysize; y++) {
      if (grid.mapm[z * xsize * ysize + y * xsize + x] == 0) {
      color[x][y] = 0;
      }
      else {
      color[x][y] = num_colors;
      num_colors++;
      }
    }
  }

  // Use spreading activation to segment regions

  do {
    changed = 0;
    for (x = 0; x < xsize; x++) {
      for (y = 0; y < ysize; y++) {
      for (nx = x - 1; nx <= x + 1; nx++) {
        for (ny = y - 1; ny <= y + 1; ny++) {
          if ((nx >= 0) && (nx < GLOBAL_X_RES) &&
            (ny >= 0) && (ny < GLOBAL_Y_RES)) {
```

```
              if ((color[nx][ny] > 0) && (color[nx][ny] < color[x][y])) {
              color[x][y] = color[nx][ny];
              changed = 1;
              }
            }
          }
        }
        }
      }
    }
    while(changed);
  }

  void agent::print_region_map(int grid[GLOBAL_X_RES][GLOBAL_Y_RES])
                          // Colored grid
  {
    // Print colored grid cell values

    char symbol;                 // Cell symbol
    int x, y;                // Cell index

    cout << endl;
    for (x = 0; x < GLOBAL_X_RES;x++) {
      for (y = 0; y < GLOBAL_Y_RES; y++) {
        if (grid[x][y] == 0) {
        cout << ".";
        }
        else {
        if (grid[x][y] < 10) {
          symbol = '0' + (char) grid[x][y];
        }
        else if (grid[x][y] < 36) {
          symbol = 'A' + (char) (grid[x][y] - 10);
        }
        else {
          symbol = 'a' + (char) (grid[x][y] - 36);
        }
        cout << symbol;
        }
      }
      cout << endl;
    }
    cout << endl;
  }

  void agent::analyze_regions(int grid[GLOBAL_X_RES][GLOBAL_Y_RES])
                          // Colored grid
  {
    // Determine size and centroid of frontier regions

    double xscale, yscale;             // Cell dimensions (tenths of
  inches)
    double cx, cy;               // Centroid of new region
    int count[MAX_COLORS];             // Edge cell counter for regions
    int x_sum[MAX_COLORS];             // Sum of cell x-coords
    int y_sum[MAX_COLORS];             // Sum of cell y-coords
    int x, y;                    // Cell index
    int i;
```

```

  num_front = 0;

  xscale = (GLOBAL_X_MAX - GLOBAL_X_MIN) * 120.0 / (double)
GLOBAL_X_RES;
  yscale = (GLOBAL_Y_MAX - GLOBAL_Y_MIN) * 120.0 / (double)
GLOBAL_Y_RES;

  for (i = 0; i < MAX_COLORS; i++) {
    count[ i] = 0;
    x_sum[ i] = 0;
    y_sum[ i] = 0;
  }

  for (x = 0; x < GLOBAL_X_RES; x++) {
    for (y = 0; y < GLOBAL_Y_RES; y++) {
      if (grid[ x][ y] > 0) {
      count[ grid[ x][ y]]++;
      x_sum[ grid[ x][ y]] += x;
      y_sum[ grid[ x][ y]] += y;
      }
    }
  }

  for (i = 1; i < MAX_COLORS; i++) {
    if (count[ i] >= MIN_REGION_SIZE) {
      cx =
      xscale * (double) x_sum[ i] / (double) count[ i] + GLOBAL_X_MIN *
120.0;
      cy =
      yscale * (double) y_sum[ i] / (double) count[ i] + GLOBAL_Y_MIN *
120.0;

      if (!(visited(cx, cy) || inaccessible(cx, cy))) {
      //      if (!inaccessible(cx, cy)) {
      if (num_front == MAX_FRONTIERS) {
        cout << "analyze_regions: Too many regions (>" << MAX_FRONTIERS
             << ")." << endl;
      }
      else {
        frontiers[ num_front].color = i;
        frontiers[ num_front].size = count[ i];
        frontiers[ num_front].x = cx;
        frontiers[ num_front].y = cy;
        num_front++;
      }
      }
    }
  }

  for (i = 0; i < num_front; i++) {
    cout << "Region [" << i << "] : size = " << frontiers[ i].size
       << " : centroid = (" << frontiers[ i].x << ", " << frontiers[ i].y
       << ")" << endl;
  }
}

int agent::visited(double cx, double cy)  // Centroid of new region
```

```
{
  // Check whether centroid corresponds to previously visited frontier
  // Return 1 if visited, 0 otherwise

  double dist;          // Distance from new region to visited frontier
  int i;

  //  cout << "Checking (" << cx << ", " << cy << ") against visited
list."
  //       << endl;

  for (i = 0; i < num_visit; i++) {
    dist = hypot(cx - front_visit[i].x, cy - front_visit[i].y);
    //    cout << "front_visit[" << i << "] at (" << front_visit[i].x <<
", "
    //       << front_visit[i].y << ") : distance = " << dist;
    if (dist <= VISIT_RADIUS) {
      //      cout << " [- VISITED -]" << endl;
      return(1);
    }
    //    cout << endl;
  }

  return(0);
}

int agent::inaccessible(double cx, double cy)   // Centroid of new
region
{
  // Check whether centroid corresponds to inaccessible frontier
  // Return 1 if inaccessible, 0 otherwise

  double dist;          // Distance from new region to inaccessible
frontier
  int i;

  //  cout << "Checking (" << cx << ", " << cy << ") against
inaccessible list."
  //       << endl;

  for (i = 0; i < num_inac; i++) {
    dist = hypot(cx - front_inac[i].x, cy - front_inac[i].y);
    //    cout << "front_inac[" << i << "] at (" << front_inac[i].x <<
", "
    //       << front_inac[i].y << ") : distance = " << dist;
    if (dist <= INAC_RADIUS) {
      //      cout << " [* INACCESSIBLE *]" << endl;
      return(1);
    }
    //    cout << endl;
  }

  return(0);
}

int agent::closest_frontier(double x, double y)
{
  // Return index of unvisited, accessible frontier closest to (x, y)
```

```
  // Return -1 if no such frontier exists

  double min_dist = MAX_DIST; // Minimum distance to frontier
  double dist = -1;           // Distance to frontier
  int close_index = -1;       // Index of closest frontier
  int i;

  for (i = 0; i < num_front; i++) {
    if (!(visited(frontiers[i].x, frontiers[i].y) ||
        inaccessible(frontiers[i].x, frontiers[i].y))) {
      //    if (!inaccessible(frontiers[i].x, frontiers[i].y)) {
      dist = hypot(x - frontiers[i].x, y - frontiers[i].y);
      if (dist < min_dist) {
      min_dist = dist;
      close_index = i;
      }
    }
  }

  return(close_index);
}

void agent::display_region_centroids(double cx, // Display center x-
coord
                             double cy)   // Display center y-coord
{
  // Mark region centroids in evidence grid window

  double xd, yd;              // Display coords
  char label[STRLEN];               // Mark label (index)
  int mark_color;             // Mark color
  int i;

  for (i = 0; i < num_front; i++) {
    xd = frontiers[i].x - cx;
    yd = frontiers[i].y - cy;

    mark_color = (frontiers[i].color - 1) % DISPLAY_COLORS;
    //    cout << "Drawing frontier [" << i << "] in " <<
color_table[mark_color]
    //       << " (" << mark_color << ")" << endl;

    global_window->set_color(color_table[mark_color]);
    global_window->display_circle(xd, yd, CENTROID_MARK_RADIUS);
    global_window->display_line(xd - CENTROID_MARK_RADIUS, yd,
                         xd + CENTROID_MARK_RADIUS, yd);
    global_window->display_line(xd, yd - CENTROID_MARK_RADIUS,
                         xd, yd + CENTROID_MARK_RADIUS);

    sprintf(label, "%d", i);
    global_window->display_text(xd + CENTROID_MARK_RADIUS * 2.0, yd,
label);
    global_window->set_color("black");
  }
  global_window->flush();
  //  cout << endl;

  //  for (i = 0; i < DISPLAY_COLORS; i++) {
```

```
  //    global_window->set_color(color_table[ i] );
  //    global_window->display_line(0, i * -100, 1000, i * -100);
  //  }
  //  global_window->set_color("black");
  //  global_window->flush();
}

void agent::display_robot_region_centroids(void)
{
  // Mark region centroids in robot window

  int xd, yd;                        // Display coords
  int mark_color;                 // Mark color
  int color_mode;                 // Color mode for draw command
  int i;

  refresh_all();

  for (i = 0; i < num_front; i++) {
    xd = (int) frontiers[ i] .x;
    yd = (int) frontiers[ i] .y;

    //    mark_color = (frontiers[ i] .color - 1) % DISPLAY_COLORS;
    //    color_mode = robot_color[ mark_color]  + 2;
    //    cout << "Drawing frontier [" << i << "] in " <<
color_table[ mark_color]
    //       << " (" << robot_color[ mark_color] << ") [ mode " <<
color_mode << "]"
    //       << endl;

    color_mode = 1;

    draw_arc(xd - (int) CENTROID_MARK_RADIUS, yd + (int)
CENTROID_MARK_RADIUS,
             (int) (CENTROID_MARK_RADIUS * 2.0),
             (int) (CENTROID_MARK_RADIUS * 2.0),
             0, 3600, color_mode);
    draw_line(xd - (int) CENTROID_MARK_RADIUS, yd,
              xd + (int) CENTROID_MARK_RADIUS, yd, color_mode);
    draw_line(xd, yd - (int) CENTROID_MARK_RADIUS,
              xd, yd + (int) CENTROID_MARK_RADIUS, color_mode);
  }
  //  cout << endl;

  //  for (i = 0; i < DISPLAY_COLORS; i++) {
  //    color_mode = robot_color[ i]  + 2;
  //    draw_line(0, i * -100, 1000, i * -100, color_mode);
  //  }
}

int agent::check_frontier_cell(int x, int y,    // Cell index
                         int front_index) // Frontier index
{
  // Check whether cell (x, y) is part of frontier <front_index>

  if (frontiers[ front_index] .color == region_map[ x][ y] ) {
    return(1);
  }
```

```
  else {
    return(0);
  }
}

/******************** NAVIGATION FUNCTIONS ********************/

void agent::corridor_advance(void)
{
  // Move forward if front corridor is clear

  if (wide_corridor[FWD] == 1) {
// TEMP FIX for SCOUT comment out mv command below and change to
scout_vm
//    mv(MV_VM, CORRIDOR_SPEED_WIDE, MV_IGNORE, 0, MV_IGNORE, 0);
cout << "In corridor_advance wide_corridor about to call scout_vm ("
        << CORRIDOR_SPEED_WIDE << ", 0" << endl;  // TEMP FIX for SCOUT
    scout_vm(CORRIDOR_SPEED_WIDE, 0);  // TEMP FIX for SCOUT
  }
  else if (corridor[FWD] == 1) {
// TEMP FIX for SCOUT comment out mv command below and change to
scout_vm
//    mv(MV_VM, CORRIDOR_SPEED, MV_IGNORE, 0, MV_IGNORE, 0);
cout << "In corridor_advance corridor about to call scout_vm ("
        << CORRIDOR_SPEED << ", 0" << endl;  // TEMP FIX for SCOUT
    scout_vm(CORRIDOR_SPEED, 0);  // TEMP FIX for SCOUT
  }
  else {
// TEMP FIX for SCOUT comment out mv command below and change to
scout_vm
//    mv(MV_VM, 0, MV_IGNORE, 0, MV_IGNORE, 0);
cout << "In corridor_advance else about to call scout_vm (0,0" << endl;
// TEMP FIX for SCOUT
    scout_vm(0, 0);  // TEMP FIX for SCOUT
  }
}

void agent::goal_corridor_orient(int gx, int gy)
{
    // Turn toward clear corridor closest to goal bearing

    double bearing;           // Bearing from robot to goal
    double corridor_index;    // Index of selected corridor (-1 = none)
    double corridor_bearing;  // Bearing of selected corridor
    double cmd_bearing;       // Bearing to face

    update();

    bearing = atan2((double) (gy - r.y), (double) (gx - r.x)) * RAD2DEG;

    //    cout << "goal = (" << gx << ", " << gy << ") : current = (" <<
r.x << ", "
    //        << r.y << ") : distance = "
    //        << hypot((double) (gy - r.y), (double) (gx - r.x)) / 10.0
    //          << " : bearing = " << bearing << endl;

    detect_corridors();
    corridor_index = select_corridor(bearing);
```

```
    corridor_bearing =
      angle_wrap((double) (corridor_index * SENSOR_SEP + r.theta) /
10.0);

    if ((corridor_index == -1) ||
      (angle_diff(bearing, corridor_bearing) > CORRIDOR_MAX_DEVIATION))
{
      cmd_bearing =
      angle_wrap((double) r.theta / 10.0 +
                 rdrand(-GOAL_CORRIDOR_NOISE, GOAL_CORRIDOR_NOISE));
    }
    else {
      cmd_bearing =
      angle_wrap(corridor_bearing +
                 rdrand(-GOAL_CORRIDOR_NOISE, GOAL_CORRIDOR_NOISE));
    }

    //    cout << "corridor index = " << corridor_index << " : corridor
bearing = "
    //       << corridor_bearing << " : command bearing = " <<
cmd_bearing << endl;

    r.face_angle_fast((int) (cmd_bearing * 10.0));   // TEMP FIX for
SCOUT
}

void agent::update_nav_grid(void)
{
  // Update navigation grid based on global grid

  // grid_fine_to_coarse(global_grid, nav_grid);

  grid_copy(nav_grid, global_grid);

  //  grid_display(nav_window, nav_grid);
}

int agent::path_plan(double wx, double wy,      // World coords of goal
                     path &nav_path)         // Navigation path (optimized)
{
  // Plan path to goal location (return 1 if path found, 0 otherwise)

  path rev_path;  // Reversed path
  path unopt_path;      // Unoptimized navigation path
  path opt_path;  // Optimized navigation path
  int gx, gy, gz; // Grid coordinates of destination
  int rx, ry, rz; // Grid coordinates of robot
  int x, y;       // Cell index
  int search_status;    // Flag indicating whether path has been found

  world2grid(nav_grid, (double) r.x / 120.0, (double) r.y / 120.0,
             0, &rx, &ry, &rz);

  world2grid(nav_grid, wx / 120.0, wy / 120.0, 0, &gx, &gy, &gz);

  cout << "Robot location = (" << r.x << ", " << r.y << ") [" << rx <<
", "
       << ry << "]" << endl;
```

```
  cout << "Goal location = (" << wx << ", " << wy << ") [" << gx << ", "
       << gy << "]" << endl;

  update_nav_grid();

  for (x = 0; x < NAV_X_RES; x++) {
    for (y = 0; y < NAV_Y_RES; y++) {
      visit[x][y] = 0;
    }
  }

  search_status = find_path(rx, ry, gx, gy, rev_path);

  if (search_status == SEARCH_FAIL) {
    cout << "No path found." << endl;
    return(0);
  }

  reverse_path(rev_path, unopt_path);

  cout << "Unoptimized: ";
  print_path(unopt_path);

  optimize_path(unopt_path, opt_path);
  cout << "Optimized: ";
  print_path(opt_path);

  generate_world_path(opt_path, nav_path);
  cout << "World path:";
  print_path(nav_path);

  //  grid_display(nav_window, nav_grid);
  //  display_path(nav_path, OPT_PATH_COLOR, nav_window);
  display_path(nav_path, OPT_PATH_COLOR, global_window);
  //  display_path_robot(nav_path, ROBOT_OPT_PATH_COLOR);

  return(1);
}

int agent::frontier_path_plan(double wx, double wy, // World coords of
goal
                        int front_index,      // Frontier index
                        path &nav_path)       // Navigation path
{
  // Plan path to goal location (return 1 if path found, 0 otherwise)

  path rev_path;  // Reversed path
  path unopt_path;      // Unoptimized navigation path
  path opt_path;  // Optimized navigation path
  int gx, gy, gz; // Grid coordinates of destination
  int rx, ry, rz; // Grid coordinates of robot
  int x, y;       // Cell index
  int search_status;    // Flag indicating whether path has been found

  world2grid(nav_grid, (double) r.x / 120.0, (double) r.y / 120.0,
          0, &rx, &ry, &rz);

  world2grid(nav_grid, wx / 120.0, wy / 120.0, 0, &gx, &gy, &gz);
```

```
  cout << "Robot location = (" << r.x << ", " << r.y << ") [" << rx <<
", "
       << ry << "]" << endl;
  cout << "Goal location = (" << wx << ", " << wy << ") [" << gx << ", "
       << gy << "]" << endl;

  update_nav_grid();

  for (x = 0; x < NAV_X_RES; x++) {
    for (y = 0; y < NAV_Y_RES; y++) {
      visit[x][y] = 0;
    }
  }

  search_status = frontier_find_path(rx, ry, gx, gy, front_index,
rev_path);

  if ((search_status == SEARCH_FAIL) || (search_status ==
SEARCH_TIMEOUT)) {
    cout << "No path found." << endl;
    return(0);
  }

  reverse_path(rev_path, unopt_path);

  cout << "Unoptimized: ";
  print_path(unopt_path);

  optimize_path(unopt_path, opt_path);
  cout << "Optimized: ";
  print_path(opt_path);

  generate_world_path(opt_path, nav_path);
  cout << "World path:";
  print_path(nav_path);

  //  grid_display(nav_window, nav_grid);
  //  display_path(nav_path, OPT_PATH_COLOR, nav_window);
  display_path(nav_path, OPT_PATH_COLOR, global_window);
  //  display_path_robot(nav_path, ROBOT_OPT_PATH_COLOR);

  return(1);
}

void agent::print_path(path p)
{
  // Print all cells on path

  int i;

  cout << "path length = " << p.length << " : path = ";

  for (i = 0; i < p.length; i++) {
    cout << "(" << p.x[i] << ", " << p.y[i] << ") ";
  }

  cout << endl;
```

```
}

void agent::display_path(path p,          // Path
                         char *pcolor,    // Path color
                         window *win)     // Window
{
  // Draw path in window

  int i;

  win->set_color(pcolor);

  for (i = 0; i < p.length - 1; i++) {
      win->display_line(p.x[ i] , p.y[ i] , p.x[ i + 1] , p.y[ i + 1] );
  }

  win->flush();
  win->set_color("black");
}

void agent::display_path_robot(path p,                // Path
                               int pcolor)            // Path color
{
  // Draw path in robot window

  int i;

  for (i = 0; i < p.length - 1; i++) {
      draw_line(p.x[ i] , p.y[ i] , p.x[ i + 1] , p.y[ i + 1] , pcolor + 2);
  }
}

int agent::find_path(int sx, int sy,      // Start cell
                     int gx, int gy,      // Goal cell
                     path &p)             // Path
{
  // Find path from (sx, sy) to (gx, gy)

  int nx, ny;            // Neighbor cell index

  path_init(p);

  visit[ sx][ sy] = 1;

  while(closest_neighbor(sx, sy, gx, gy, nx, ny)) {
    if (search_cell(nx, ny, gx, gy, p) == SEARCH_SUCCESS) {
      //      cout << "[ ON PATH (" << x << ", " << y << ") ]" << endl;
      path_add(p, sx, sy);
      return(SEARCH_SUCCESS);
    }
  }

  return(SEARCH_FAIL);
}

int agent::frontier_find_path(int sx, int sy,   // Start cell
                              int gx, int gy,   // Goal cell
                              int front_index,  // Frontier index
```

```
                          path &p)          // Path
{
  // Find path from (sx, sy) to (gx, gy) or any point on frontier
<front_index>

  int nx, ny;            // Neighbor cell index
  int status;            // Cell search status

  path_init(p);

  visit[ sx][ sy] = 1;

  while(closest_neighbor(sx, sy, gx, gy, nx, ny)) {
    cell_count = 0;
    status = frontier_search_cell(nx, ny, gx, gy, front_index, p);
    if (status == SEARCH_SUCCESS) {
      //      cout << "[ ON PATH (" << x << ", " << y << ") ]" << endl;
      path_add(p, sx, sy);
      return(SEARCH_SUCCESS);
    }
    if (status == SEARCH_TIMEOUT) {
      return(SEARCH_TIMEOUT);
    }
  }

  return(SEARCH_FAIL);
}

int agent::search_cell(int x, int y,      // Search cell
                  int gx, int gy,  // Goal cell
                  path &p)         // Path
{
  // Search cell (x,y) and return search status

  int status;                 // Search status
  int nx, ny;                 // Neighbor cell index

  if (visit[ x][ y] ) {
    cout << "search_cell: Error: revisited cell (" << x << ", " << y <<
")"
        << endl;
    exit(-1);
  }
  visit[ x][ y] = 1;

  //  cout << "Searching (" << x << ", " << y << ") : ";

  if ((x < 0) || (x >= NAV_X_RES) || (y < 0) || (y >= NAV_Y_RES)) {
    //    cout << "Out of bounds." << endl;
    return(SEARCH_FAIL);
  }


  if ((x == gx) && (y == gy)) {
    //    cout << "[* GOAL (" << x << ", " << y << ") *]" << endl;
    path_add(p, x, y);
    return(SEARCH_SUCCESS);
  }
```

```
  if ((nav_grid.mapm[ grid2index(nav_grid, x, y, 0)] >= 0) ||
      (!check_clear(x, y))) {
    //    cout << "> BLOCKED <" << endl;
    return(SEARCH_FAIL);
  }

  //  cout << "(( Searching adjacent ))" << endl;

  while(closest_neighbor(x, y, gx, gy, nx, ny)) {
    if (search_cell(nx, ny, gx, gy, p) == SEARCH_SUCCESS) {
      //      cout << "[  ON PATH (" << x << ", " << y << ") ]" << endl;
      path_add(p, x, y);
      return(SEARCH_SUCCESS);
    }
  }

  return(SEARCH_FAIL);
}

int agent::frontier_search_cell(int x, int y,          // Search cell
                            int gx, int gy,          // Goal cell
                            int front_index,  // Frontier index
                            path &p)          // Path
{
  // Search cell (x,y) while navigating to frontier and return search
status

  int child_status;            // Search status for child cell
  int nx, ny;                  // Neighbor cell index

  cell_count++;
  if (cell_count % 100 == 0) {
    cout << "Searching " << cell_count << " cells..." << endl;
  }
  if (cell_count > SEARCH_MAX_CELLS) {
    cell_count = 0;
    return(SEARCH_TIMEOUT);
  }

  if (visit[ x][ y] ) {
    cout << "frontier_search_cell: Error: revisited cell (" << x << ", "
<< y << ")"
       << endl;
    exit(-1);
  }
  visit[ x][ y] = 1;

  //  cout << "Searching (" << x << ", " << y << ") : ";

  if ((x < 0) || (x >= NAV_X_RES) || (y < 0) || (y >= NAV_Y_RES)) {
    //    cout << "Out of bounds." << endl;

    return(SEARCH_FAIL);
  }


  //  if ((x == gx) && (y == gy)) {
```

```

  if (((x == gx) && (y == gy)) ||
      (check_frontier_arrival(x, y, front_index))) {
    //    cout << "[* GOAL (" << x << ", " << y << ") *]" << endl;
    path_add(p, x, y);
    return(SEARCH_SUCCESS);
  }

  if ((nav_grid.mapm[grid2index(nav_grid, x, y, 0)] >= 0) ||
      (!check_clear(x, y))) {
    //    cout << "> BLOCKED <" << endl;
    return(SEARCH_FAIL);
  }

  //  cout << "(( Searching adjacent ))" << endl;

  while(closest_neighbor(x, y, gx, gy, nx, ny)) {
    child_status = frontier_search_cell(nx, ny, gx, gy, front_index, p);
    if (child_status == SEARCH_SUCCESS) {
      //      cout << "[ ON PATH (" << x << ", " << y << ") ]" << endl;
      path_add(p, x, y);
      return(SEARCH_SUCCESS);
    }
    if (child_status == SEARCH_TIMEOUT) {
     return(SEARCH_TIMEOUT);
    }
  }

  return(SEARCH_FAIL);
}

int agent::closest_neighbor(int x, int y,       // Current cell index
                       int gx, int gy,           // Goal cell index
                       int &nx, int &ny)         // Next cell index
{
  // Find index of (unvisited) neighbor closest to goal

  double min_dist;      // Minimum distance from neighbor to goal
  double dist;          // Distance from neighbor to goal
  int found = 0;  // 1 if unvisited neighbor exists, 0 otherwise
  int dx, dy;           // Adjacent cell offset
  int ax, ay;           // Adjacent cell index

  min_dist = (double) MAX_PATH_LENGTH;

  for (dx = -1; dx <= 1; dx++) {
    for (dy = -1; dy <= 1; dy++) {
      ax = x + dx;
      ay = y + dy;

      if ((ax >= 0) && (ax < NAV_X_RES) && (ay >= 0) && (ay <
NAV_Y_RES)) {
      if (visit[ax][ay] == 0) {
        dist = hypot(gx - ax, gy - ay);
        if (dist < min_dist) {
          min_dist = dist;
          nx = ax;
          ny = ay;
```

```
            found = 1;
          }
        }
        }
      }
    }

    return(found);
}

void agent::reverse_path(path old_path,          // Initial path
                         path &new_path)  // Reversed path
{
  // Reverse order of steps on path

  int i;

  path_init(new_path);
  for (i = 0; i < old_path.length; i++) {
    path_add(new_path, old_path.x[ (old_path.length - 1) - i],
           old_path.y[ (old_path.length - 1) - i]);
  }
}

void agent::optimize_path(path old_path,  // Initial path
                          path &new_path) // Optimized path
{
  // Optimize path by jumping between adjacent path cells

  int marker = 0; // Point along old path
  int jump_marker;      // Point to jump to on new path
  int jump_flag;  // Set to 1 if path jumps

  path_init(new_path);
  path_add(new_path, old_path.x[ 0], old_path.y[ 0]);

  //  cout << "Starting at (" << new_path.x[ 0] << ", " << new_path.y[ 0]
  //       << ") [ 0] <0>" << endl;

  while(marker < old_path.length - 1) {
    jump_flag = 0;
    //    cout << "Trying to jump from (" << new_path.x[ new_path.length
- 1] << ", "
    //       << new_path.y[ new_path.length - 1] << ") [" <<
new_path.length - 1
    //       << "] <" << marker << ">" << endl;
    for (jump_marker = old_path.length - 1;
       (jump_marker > marker) && !jump_flag;
       jump_marker--) {
      //      cout << "Checking (" << old_path.x[ jump_marker] << ", "
      //       << old_path.y[ jump_marker] << ") <" << jump_marker << ">"
<< endl;
      if ((old_path.x[ jump_marker] - old_path.x[ marker] >= -1) &&
        (old_path.x[ jump_marker] - old_path.x[ marker] <= 1) &&
        (old_path.y[ jump_marker] - old_path.y[ marker] >= -1) &&
        (old_path.y[ jump_marker] - old_path.y[ marker] <= 1)) {

```

```
        path_add(new_path, old_path.x[ jump_marker] ,
old_path.y[ jump_marker] );
        //    cout << "Jumping from (" << new_path.x[ new_path.length - 2]
<< ", "
        //         << new_path.y[ new_path.length - 2] << ") to ("
        //         << new_path.x[ new_path.length - 1] << ", "
        //         << new_path.y[ new_path.length - 1] << ")" << endl;

        marker = jump_marker;
        jump_flag = 1;
        }
      }
    }
}

void agent::generate_world_path(path grid_path,        // Path in nav
grid
                        path &world_path) // Path in world coords
{
  // Convert path in grid cell coords to world coords

  double wx, wy;                // World coords
  double xscale, yscale, zscale;    // Cell dimensions (tenths of
inches)
  int xsize, ysize, zsize;          // Grid dimensions (# cells)
  int i;

  xsize = nav_grid.msize[ 0] ;
  ysize = nav_grid.msize[ 1] ;
  zsize = nav_grid.msize[ 2] ;

  xscale = (nav_grid.himv[ 0]  - nav_grid.lomv[ 0] ) * 120.0 / (double)
xsize;
  yscale = (nav_grid.himv[ 1]  - nav_grid.lomv[ 1] ) * 120.0 / (double)
ysize;
  zscale = (nav_grid.himv[ 2]  - nav_grid.lomv[ 2] ) * 120.0 / (double)
zsize;

  path_init(world_path);

  for (i = 0; i < grid_path.length; i++) {
      wx = (((double) grid_path.x[ i]  + 0.5) * xscale
          + nav_grid.lomv[ 0]  * 120.0);
      wy = (((double) grid_path.y[ i]  + 0.5) * yscale
          + nav_grid.lomv[ 1]  * 120.0);

      path_add(world_path, (int) wx, (int) wy);
  }
}

void agent::path_init(path &p)      // Path
{
  // Initialize path

  p.length = 0;
}

void agent::path_add(path &p, // Path
```

```
                    int x, int y)      // Point to add to path
{
  // Add point to path

  if (p.length == MAX_PATH_LENGTH) {
    cout << "path_add: Too many steps (> " << MAX_PATH_LENGTH << ")." <<
endl;
    exit(-1);
  }

  p.x[p.length] = x;
  p.y[p.length] = y;
  p.length++;
}

int agent::check_clear(int x, int y)
{
  //  Check to see whether region around point is free of known
obstacles

  int obs_count = 0;            // Obstacle counter
  int xi, yi;                   // Grid indices
  int xl, xh, yl, yh, zl, zh; // Grid coords of robot-sized box around
point
  double wx, wy, wz;            // World coordinates of point
  double wxl, wxh, wyl, wyh;  // World coords of robot-sized box around
point

  //  int xsize, ysize;         // Grid dimensions (# cells)
  //  double xscale, yscale;  // Cell dimensions (tenths of inches)
  //  double xd, yd;            // Display coords

  grid2world(nav_grid, x, y, 0, &wx, &wy, &wz);

  wxl = wx - ROBOT_PASSAGE_RADIUS;
  wxh = wx + ROBOT_PASSAGE_RADIUS;
  wyl = wy - ROBOT_PASSAGE_RADIUS;
  wyh = wy + ROBOT_PASSAGE_RADIUS;

  world2grid(nav_grid, wxl, wyl, wz, &xl, &yl, &zl);
  world2grid(nav_grid, wxh, wyh, wz, &xh, &yh, &zh);

  //  xsize = NAV_X_RES;
  //  ysize = NAV_Y_RES;

  //  xscale = (NAV_X_MAX - NAV_X_MIN) * 120.0 / (double) xsize;
  //  yscale = (NAV_Y_MAX - NAV_Y_MIN) * 120.0 / (double) ysize;

  for (xi = xl; xi <= xh; xi++) {
    for (yi = yl; yi <= yh; yi++) {
      //      xd = (double) (xi + 0.5) * xscale + GLOBAL_X_MIN * 120.0;
      //      yd = (double) (yi + 0.5) * yscale + GLOBAL_Y_MIN * 120.0;

      //      global_window->set_color("gold");
      //      global_window->display_point(xd, yd);
      //      global_window->set_color("black");

      if (nav_grid.mapm[grid2index(nav_grid, xi, yi, 0)] > 0) {
```

```
      //    global_window->set_color("red");
      //    global_window->display_circle(xd, yd, xscale);
      //    global_window->set_color("black");
      obs_count++;
      }
    }
  }

  if (obs_count > MAX_OBS_COUNT) {
    return(0);
  }
  else {
    return(1);
  }
}

int agent::check_frontier_arrival(int x, int y, int front_index)
{
  //  Check to see whether region around point overlaps frontier

  int xi, yi;                 // Grid indices
  int xl, xh, yl, yh, zl, zh; // Grid coords of robot-sized box around
point
  double wx, wy, wz;          // World coordinates of point
  double wxl, wxh, wyl, wyh;  // World coords of robot-sized box around
point

  grid2world(nav_grid, x, y, 0, &wx, &wy, &wz);

  wxl = wx - ROBOT_PASSAGE_RADIUS;
  wxh = wx + ROBOT_PASSAGE_RADIUS;
  wyl = wy - ROBOT_PASSAGE_RADIUS;
  wyh = wy + ROBOT_PASSAGE_RADIUS;

  world2grid(nav_grid, wxl, wyl, wz, &xl, &yl, &zl);
  world2grid(nav_grid, wxh, wyh, wz, &xh, &yh, &zh);

  for (xi = xl; xi <= xh; xi++) {
    for (yi = yl; yi <= yh; yi++) {
      if (check_frontier_cell(xi, yi, front_index)) {
      return(1);
      }
    }
  }
  return(0);
}

double agent::closest_waypoint(path p,          // Path
                        int x, int y,    // Current position (1/10
inch)
                        int index, // Index of current waypoint
                        int &close_index) // Index of closest waypoint
{
  // Finds waypoint furthest on path within destination tolerance, or
  // waypoint on path <p> closest to (x, y), returning the distance
(inches)
  // to that point, and the waypoint's index in <index>

```

```
  double dist;                  // Distance to waypoint
  double min_dist = MAX_DIST; // Minimum distance to waypoint
  int last_waypoint;            // Last waypoint to check
  int i;

  //  cout << "current position = (" << x << ", " << y << ")" << endl;

  if ((index < 0) || (index >= p.length)) {
    cout << "closest_waypoint: index <" << index << "> out of range
[0.."
        << p.length << "]" << endl;
    exit(-1);
  }

  // Set lookahead window for checking waypoints

  last_waypoint = index + WAYPOINT_WINDOW;
  if (last_waypoint > p.length - 1) {
    last_waypoint = p.length - 1;
  }

  // Search for closest waypoint

  for (i = last_waypoint; i >= index; i--) {
    dist = hypot((double) (p.x[i] - x), (double) (p.y[i] - y)) / 10.0;

    //    cout << "distance to waypoint [" << i << "] (" << p.x[i] << ",
"
    //       << p.y[i] << ") = " << dist << endl;

    if (dist < min_dist) {
      min_dist = dist;
      close_index = i;

      if (min_dist <= LOCAL_NAV_TOLERANCE) {
      cout << "[* ARRIVED *]" << endl;
      return(min_dist);
      }
    }
  }

  //  cout << "closest waypoint [" << close_index << "] (" <<
p.x[close_index]
  //       << " , " << p.y[close_index] << ") : dist = " << min_dist <<
endl;

  return(min_dist);
}


/******************* CORRIDOR FUNCTIONS *******************/

void agent::detect_corridors(void)
{
  // Detect freespace cooridors in vicinity of robot

  int i;

```

```
  update();

  for (i = 0; i < NUM_RANGE; i++) {
    corridor[ i] = check_corridor(i, CORRIDOR_FWD_RANGE,
                                   CORRIDOR_SIDE_CLEARANCE);
    wide_corridor[ i] = check_corridor(i, CORRIDOR_WIDE_FWD_RANGE,
                                        CORRIDOR_WIDE_SIDE_CLEARANCE);
  }

  //  display_corridors();
  //  global_window->flush();

  //  display_corridors();
}


int agent::check_corridor(int center,     // Index of sensor in center
of corridor
                    int fwd_range,  // Required forward space
                    int side_clear) // Required side space
{
  // Check whether a corridor exists centered around sensor <center>
  // Return 1 if true, 0 otherwise

  int sensor;           // Sensor index
  int start;            // First sensor to be checked
  int end;         // Last sensor to be checked

// cout << "Checking corridor [" << center << "]" << endl;  // TEMP FIX
for SCOUT

  start = wrap(center - CORRIDOR_SPAN, 0, NUM_RANGE - 1);
  end = wrap(center + CORRIDOR_SPAN, 0, NUM_RANGE - 1);

  if (start < end) {
    for (sensor = start; sensor <= end; sensor++) {
      if (!corridor_check_sensor(center, sensor, fwd_range, side_clear))
{
// cout << "Corridor [" << center << "] is >> BLOCKED <<." << endl;  //
TEMP FIX for SCOUT
      return(0);
      }
    }
// cout << "Corridor [" << center << "] is [[ OPEN ]]." << endl;   //
TEMP FIX for SCOUT
    return(1);
  }

  for (sensor = start; sensor < NUM_RANGE; sensor++) {
    if (!corridor_check_sensor(center, sensor, fwd_range, side_clear)) {
// cout << "Corridor [" << center << "] is >> BLOCKED <<." << endl; //
TEMP FIX for SCOUT
      return(0);
    }
  }

  for (sensor = 0; sensor <= end; sensor++) {
    if (!corridor_check_sensor(center, sensor, fwd_range, side_clear)) {
```

```
// cout << "Corridor [" << center << "] is >> BLOCKED <<." << endl;  //
TEMP FIX for SCOUT
      return(0);
    }
  }
// cout << "Corridor [" << center << "] is [[ OPEN ]]." << endl;  //
TEMP FIX for SCOUT
  return(1);
}

int agent::corridor_check_sensor(int center,          // Center sensor
index
                          int sensor,             // Sensor index
                          int fwd_range,          // Required fwd space
                          int side_clear)  // Required side space
{
  // Check whether <sensor> is clear for corridor <center>

  const double bot_width = ROBOT_RADIUS * 12.0; // Robot width (inches)
  const double sens_sep = SENSOR_SEP * 0.1;     // Sensor separation
(degrees)

  double angle;                // Sensor angle (degrees)
  double center_angle;         // Angle of center sensor (degrees)
  double theta;                // Angle (degrees) between sensor and
                         // perpendicular to center angle
  double thresh;         // Minimum clear distance (inches)

  center_angle = angle_wrap((double) r.theta * 0.1
                      + (double) center * sens_sep);
// TEMP FIX for SCOUT - changed r.turret to r.theta on previous line
// cout << "center [" << center << "] : center angle = " << center_angle
<< endl;  // TEMP FIX for SCOUT

  angle = angle_wrap((double) r.theta * 0.1 + (double) sensor *
sens_sep);  // TEMP FIX for SCOUT r.turret to r.theta
  theta = 90.0 - angle_diff(center_angle, angle);

  if (center == sensor) {
    thresh = fwd_range;
  }
  else {
    thresh = (bot_width + side_clear) / cos(theta * DEG2RAD)
      - bot_width;
    if (thresh > fwd_range) {
      thresh = fwd_range;
    }
  }

// cout << "sensor [" << sensor << "] : sensor angle = " << angle
//          << " : theta = " << theta << " : thresh = " << thresh
//         << " : range = " << r.range[sensor];  // TEMP FIX for SCOUT

  if (r.range[sensor] < thresh) {
// cout << " * BLOCKED *" << endl;  // TEMP FIX for SCOUT
    return(0);
  }
  else {
```

```
// cout << " [  CLEAR ]" << endl;   // TEMP FIX for SCOUT
    return(1);
  }
}

void agent::display_corridors(void)
{
  // Display corridors in robot window

  int i;

  //  refresh_all();

  for (i = 0; i < NUM_RANGE; i++) {
    if (wide_corridor[ i] == 1) {
      display_corridor(global_window, i, CORRIDOR_WIDE_FWD_RANGE,
                   CORRIDOR_WIDE_SIDE_CLEARANCE, CORRIDOR_WIDE_COLOR);
      //      display_corridor_robot(i, CORRIDOR_WIDE_FWD_RANGE,
      //                      CORRIDOR_WIDE_SIDE_CLEARANCE,
      //                      CORRIDOR_WIDE_COLOR_ROBOT);
    }
    else if (corridor[ i] == 1) {
      display_corridor(global_window, i, CORRIDOR_FWD_RANGE,
                   CORRIDOR_SIDE_CLEARANCE, CORRIDOR_COLOR);
      //      display_corridor_robot(i, CORRIDOR_FWD_RANGE,
      //                      CORRIDOR_SIDE_CLEARANCE,
CORRIDOR_COLOR_ROBOT);
    }
  }
}

void agent::display_corridor(window *win, // Window
                       int center,  // Center sensor index
                       int fwd_range,     // Required forward space
                       int side_clear,    // Required side space
                       char *color) // Corridor color
{
  // Display corridor boundaries centered around sensor <center>

  // Robot width (1/10 inch)
  const double bot_width = ROBOT_RADIUS * 120.0;

  double fwd_dist;      // Length of forward axis (1/10 inch)
  double side_dist;     // Distance to either side of robot (1/10 inch)
  double angle;                         // Corridor angle (degrees)
  double x1, y1, x2, y2, x3, y3, x4, y4;  // Corner coords (1/10 inch)
  double fwd_x, fwd_y;                  // Offset for forward end

  fwd_dist = bot_width + (double) fwd_range * 10.0;
  side_dist = bot_width + (double) side_clear * 10.0;

// SCOUT THESIS CHANGE - changed r.turret to r.theta in line below
  angle = angle_wrap((double) r.theta * 0.1
              + (double) (center * SENSOR_SEP) * 0.1);

  x1 = r.x + side_dist * cos((angle + 90.0) * DEG2RAD);
  y1 = r.y + side_dist * sin((angle + 90.0) * DEG2RAD);

```

```
  x2 = r.x + side_dist * cos((angle - 90.0) * DEG2RAD);
  y2 = r.y + side_dist * sin((angle - 90.0) * DEG2RAD);

  fwd_x = fwd_dist * cos(angle * DEG2RAD);
  fwd_y = fwd_dist * sin(angle * DEG2RAD);

  x3 = x1 + fwd_x;
  y3 = y1 + fwd_y;

  x4 = x2 + fwd_x;
  y4 = y2 + fwd_y;

  win->set_color(color);

  win->display_line(x1, y1, x2, y2);
  win->display_line(x2, y2, x4, y4);
  win->display_line(x4, y4, x3, y3);
  win->display_line(x3, y3, x1, y1);

  win->set_color("black");
}

void agent::display_corridor_robot(int center,  // Center sensor index
                           int fwd_range,  // Required forward space
                           int side_clear, // Required side space
                           int color)     // Corridor color
{
  // Display corridor boundaries centered around sensor <center> in
robot window

  // Robot width (1/10 inch)
  const double bot_width = ROBOT_RADIUS * 120.0;

  double fwd_dist;      // Length of forward axis (1/10 inch)
  double side_dist;     // Distance to either side of robot (1/10 inch)
  double angle;                             // Corridor angle (degrees)
  double x1, y1, x2, y2, x3, y3, x4, y4;  // Corner coords (1/10 inch)
  double fwd_x, fwd_y;                      // Offset for forward end

  fwd_dist = bot_width + (double) fwd_range * 10.0;
  side_dist = bot_width + (double) side_clear * 10.0;

// SCOUT THESIS CHANGE - changed r.turret to r.theta in line below
  angle = angle_wrap((double) r.theta * 0.1
               + (double) (center * SENSOR_SEP) * 0.1);

  x1 = r.x + side_dist * cos((angle + 90.0) * DEG2RAD);
  y1 = r.y + side_dist * sin((angle + 90.0) * DEG2RAD);

  x2 = r.x + side_dist * cos((angle - 90.0) * DEG2RAD);
  y2 = r.y + side_dist * sin((angle - 90.0) * DEG2RAD);

  fwd_x = fwd_dist * cos(angle * DEG2RAD);
  fwd_y = fwd_dist * sin(angle * DEG2RAD);

  x3 = x1 + fwd_x;
  y3 = y1 + fwd_y;

```

```
  x4 = x2 + fwd_x;
  y4 = y2 + fwd_y;

  draw_line((int) x1, (int) y1, (int) x2, (int) y2, color + 2);
  draw_line((int) x2, (int) y2, (int) x4, (int) y4, color + 2);
  draw_line((int) x4, (int) y4, (int) x3, (int) y3, color + 2);
  draw_line((int) x3, (int) y3, (int) x1, (int) y1, color + 2);
}

int agent::select_corridor(double heading)      // Heading (degrees)
{
  // Select corridor nearest to specified heading

  const double sens_sep = SENSOR_SEP * 0.1;     // Sensor separation
(degrees)
  double angle;                 // Sensor angle
  double dtheta;        // Angle/heading deviation
  double min_dtheta = 360.0;        // Minimum angle deviation
  int select = -1;              // Index of selected corridor
  int i;

  heading = angle_wrap(heading);

  for (i = 0; i < NUM_RANGE; i++) {
    if (corridor[ i] == 1) {
      angle = angle_wrap((double) r.theta * 0.1
                  + (double) i * sens_sep);
// SCOUT THESIS CHANGE - use r.theta vice r.turret in line above
// TEMP FIX for SCOUT - lets try some numbers checking below
// cout << "About to call angle_diff with heading= " << heading
//        << "and angle = " << angle << endl;   // TEMP FIX for SCOUT
      dtheta = angle_diff(heading, angle);
// cout << "dtheta = angle_diff(heading,angle) = " << dtheta << endl;
// TEMP FIX for SCOUT
// cout << "min_dtheta = " << min_dtheta << "  i = " << i << endl;   //
TEMP FIX for SCOUT
      if (dtheta < min_dtheta) {
      min_dtheta = dtheta;
      select = i;
      }
    }
  }

  if (select == -1) {
    cout << "No open corridors." << endl;
    return(select);
  }
//SCOUT THESIS CHANGE -  changed r.turret to r.theta 3 lines down
  cout << "desired heading = " << heading << " : selected corridor ["
       << select << "] : corridor angle = "
       << angle_wrap((double) r.theta * 0.1 + (double) select *
sens_sep)
       << " : deviation = " << min_dtheta << endl;

  if (wide_corridor[ select] == 1) {
    //    display_corridor(global_window, select,
CORRIDOR_WIDE_FWD_RANGE,
```

```
    //                 CORRIDOR_WIDE_SIDE_CLEARANCE,
CORRIDOR_SELECT_WIDE_COLOR);
    //     display_corridor_robot(select, CORRIDOR_WIDE_FWD_RANGE,
    //                      CORRIDOR_WIDE_SIDE_CLEARANCE,
    //                      CORRIDOR_SELECT_WIDE_COLOR_ROBOT);
  }
  else {
    //     display_corridor(global_window, select, CORRIDOR_FWD_RANGE,
    //                 CORRIDOR_SIDE_CLEARANCE, CORRIDOR_SELECT_COLOR);
    //     display_corridor_robot(select, CORRIDOR_FWD_RANGE,
CORRIDOR_SIDE_CLEARANCE,
    //                      CORRIDOR_SELECT_COLOR_ROBOT);
  }

  return(select);
}

/********** INTERFACE TO CONTINUOUS LOCALIZATION **********/

void agent::connect_cl(void)
{
  // Initialize communications with continuous localization

  char comm_mach[STRLEN];

  cout << "Enter continuous localization host ==> ";
  cin >> comm_mach;
  cin.get();

  //  connect_to_CL(CONTLOC_CHANNEL, CONTLOC_HOST);
  //  cout << "Connected to CONTINUOUS LOCALIZATION on " << CONTLOC_HOST
<< "."
  //       << endl;

  connect_to_CL(CONTLOC_CHANNEL, comm_mach);
  cout << "Connected to CONTINUOUS LOCALIZATION on " << comm_mach << "."
       << endl;

  contloc_mode = 1;
}

void agent::send_cl_grid(void)
{
  // Send global grid to continuous localization

  if (!contloc_mode) {
    return;
  }

  cout << "Sending global grid to CONTINUOUS LOCALIZATION." << endl;
  save_grid_file(global_grid, ARIEL_CL_FILE, "");

// SCOUT THESIS CHANGE - if continuouse localization is ever used send
r.theta instead of r.turret
  sendroom_to_CL(CONTLOC_CHANNEL, ARIEL_CL_FILE, (double) r.x / 10.0,
           (double) r.y / 10.0, (double) r.theta / 10.0,
           (double) r.theta / 10.0, 0, 0.0, 0.0, 0.0);
}
```

```
/********** MULTIROBOT COMMUNICATION **********/

// BEGIN SCOUT THESIS CHANGE
// This routine now sets up communication for up to MAX_ROBOTS
simultaneously
//  2 robot limitation is eliminated

void agent::init_robot_comm(void)
{
  // Initialize robot communication channel

  char robot_server_name[STRLEN];  // Server robot host name

  // If Server Robot
  if (r.id == SERVER_ROBOT) {
    if (init_comm_server(ARIEL_CHANNEL, PORT_ARIEL, NONBLOCK_COMM) == 0)
{
      cout << "init_robot_comm: Robot [" << r.id
         << "] initialized communications as server." << endl;
      multi_mode = 1;
    }
    else {
      cout << "init_robot_comm: Robot [" << r.id
         << "] unable to set up communications as server." << endl;
      multi_mode = 0;
    }
  }
  else if (r.id <= MAX_ROBOTS) {
    cout << "Enter host name for server robot  ==> ";
    cin >> robot_server_name;

    if (init_comm_client(ARIEL_CHANNEL, robot_server_name,
      BASE_CLIENT_PORT + r.id, NONBLOCK_COMM) == 0) {
      cout << "init_robot_comm: Robot [" << r.id
         << "] initialized communications as client." << endl;
      multi_mode = 1;
    }
    else {
      cout << "init_robot_comm: Robot [" << r.id
         << "] unable to set up communications as client." << endl;
      multi_mode = 0;
    }
  }
  else {
    cout << "init_robot_comm: Robot [" << r.id
       << "] unable to set up communications for more than "
      << MAX_ROBOTS
      <<  " robots." << endl;
    multi_mode = 0;
  }
}
//END SCOUT THESIS CHANGE

void agent::send_robot_message(char *message)
{
  // Send message to other robot
// BEGIN SCOUT THESIS CHANGE
```

```
    cout << "Sending message <" << message << ">." << endl;
    // Loop thru all possible client robots connections, send message
    // that new map is available.
    for (int i=1; i< MAX_ROBOTS; i++){
       write_comm(i, message, strlen(message) + 1);
    }
  //END SCOUT THESIS CHANGE
  }

  void agent::user_send_robot_message(void)
  {
    // Send message to other robot (user mode)

    char message[STRLEN];

    cout << "Enter message ==> ";
    cin >> message;

    cout << "Sending message <" << message << ">." << endl;

    write_comm(ARIEL_CHANNEL, message, strlen(message) + 1);
  }

  //BEGIN SCOUT THESIS CHANGE
  // Pass in the channel used for communication between client
  // and server

  int agent::receive_robot_message(int channel, char *message)
  {
    // Receive message from other robot
    // Returns 1 if message received, 0 otherwise

    int message_received;       // Message receipt flag

    message_received = read_comm(channel, message, STRLEN);
  // END SCOUT THESIS CHANGE
    if (message_received) {
      cout << "Received message <" << message << ">." << endl;
    }

    return(message_received);
  }

  void agent::user_receive_robot_message(void)
  {
    // Receive message from other robot (user mode)

    char message[STRLEN];

    if (read_comm(ARIEL_CHANNEL, message, STRLEN) == 0) {
      cout << "No messages waiting." << endl;
    }
    else {
      cout << "Received message <" << message << ">." << endl;
    }
  }

  /******************* MULTIROBOT EXPLORATION *******************/
```

```
void agent::integrate_remote_map(void)
{
  // Integrate new map from remote robot (if a new map exists)

  Map3D remote_grid;          // Evidence grid for remote map
  char mapfile[ STRLEN] ;     // Remote map file
  char posinfo[ STRLEN] ;     // Remote map position information
  int mx, my, mtheta;         // Position of center of new map
                // (1/10 inch, 1/10 degree)
// BEGIN SCOUT THESIS CHANGE
  int channel;                // Channel number

  // Loop thru all channels corresponding to client robots. Check each
  // channel to see if we received a new map message.
  // Robot 2 is on channel 1, Robot3 is on channel 2, . . .

  for (channel=1; channel < MAX_ROBOTS; channel++){

     // Check for new map message

     if (!receive_robot_message(channel, mapfile)) {
       continue;              // If nothing to read on this channel,
                 // do not give up, continue will jump
                 // back to "for" loop and increment
                 // channel counter.
     }

     cout << "New map from remote robot in <" << mapfile << ">." <<
endl;

     // Load grid along with position info

     if (!load_grid_file_com(&remote_grid, mapfile, posinfo)) {
       return;
     }

     sscanf(posinfo, "%d %d %d", &mx, &my, &mtheta);

     cout << "New map position = (" << mx << ", " << my << ") [" <<
mtheta << "]"
           << endl;

     // Display and integrate new map

     //  grid_display(grid_window, remote_grid);

// NEW MAJOR SCOUT THESIS change below
// if r.id==1 then robot is SERVER and needs to integrate a local scan
to the global map
// if r.id !=1 then robot is CLIENt and needs to intgrate the SERVER
global map that is sent

    if (r.id==1)  {
     integrate_grid(global_grid, remote_grid, (double) mx / 120.0,
                 (double) my / 120.0, (double) mtheta / 10.0);
    }
    else  {
```

```
    integrate_global_grid(global_grid, remote_grid, (double) mx /
120.0,
                     (double) my / 120.0, (double) mtheta / 10.0);
  }     // close for else r.id !=1

    grid_display_global(global_grid);

  }     // close for channel check counter
// END SCOUT THESIS CHANGE
}
```

APPENDIX I. FRONTIER-BASED EXPLORATION CODE – COMM.H

This appendix contains the header file for the communications routine that allows multiple robots to send messages to one another.

```
/* Original code written by William Adams */

/*  Modifications for increased number of robots June 1998
    for Master's Thesis work by Patrick A. Hillmeyer      */

/* include file for comm.c and all files linking to comm.o */

enum{OFF_COMM,NONBLOCK_COMM,BLOCK_COMM};

#define PORT_DETECT 65003
#define PORT_GESTURE 65004
#define PORT_AUXPORT 65005
#define PORT_CONTSERV 65006
#define PORT_SEARCH 65007
#define PORT_CONTLOC 65008
#define PORT_ARIEL 65009

/* BEGIN SCOUT THESIS CHANGE */

#define MAX_ROBOTS 9  /* This is the max number of robots that can
                         operate at one time  - change as you get
                         more robots  -  this must always be one less
                         than MAXCHANNEL  - see important note about
                         MAXCHANNEL and comm.c file.          */

#define SERVER_ROBOT 1 /*  This is the ID number (in Nserver) of the
                           robot that will act as the Server robot
                           to the other Client robots for receiving and
                           sending .eg files */

#define BASE_CLIENT_PORT 65007



/*  IMPORTANT!!  The initialization of global arrays sd, ld, and
comm_mode in the file comm.c has to match EXACTLY with MAXCHANNEL.
Example: if MAXCHANNEL is 10 then you need 10 zeros to initialize each
of the arrays mentioned above.          */

#define MAXCHANNEL  MAX_ROBOTS + 1
                 /* Any single process can communicate with this
                    many other processes.  Although the port numbers
                    on both ends of the communication must agree,
                    the channel numbers do not need to agree.
                    Within a single process, each communication
                    link must have a unique channel number. */
                 /* If this is changed, you must change the initializing
                    declaration using it in comm.c */

/* END SCOUT THESIS CHANGE */
```

APPENDIX J. FRONTIER-BASED EXPLORATION CODE – COMM.C

This appendix contains the source code for the communications routine that allows multiple robots to send messages to one another.

```
/***********************************************************************
 * comm_server.c
 * written: 11/22/95  William Adams
 * last modifed: 1/22/95 William Adams
 *
 * Set up internet communication on a single port.
 * Receive requests from ONE other process and send back
 * information.

***********************************************************************/

/*  Modifications for increased number of robots June 1998
    for Master's Thesis work by Patrick A. Hillmeyer      */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <fcntl.h>
#include <errno.h>

#include "comm.h"

enum status {NOTHING_C,HALFWAY_C,READY_C};

// BEGIN SCOUT THESIS CHANGE
// *** see notes in comm.h file concerning these next few lines

int sd[MAXCHANNEL]  = {0,0,0,0,0,0,0,0,0,0};    /* socket handle */
int ld[MAXCHANNEL]  = {0,0,0,0,0,0,0,0,0,0};
int comm_mode[MAXCHANNEL] = {0,0,0,0,0,0,0,0,0,0};

// END SCOUT THESIS CHANGE

int haveaclient = 0;

/* wait client to call, blocking */
int comm_wait_for_client(channel,control
int channel,control;
{
  if (comm_mode[channel] == NOTHING_C) {
    fprintf(stderr,
      "\nImproper call to comm_wait_for_client...use
init_comm_server.\n");
    return(5);
  }
  else if (comm_mode[channel] == READY_C) {
    fprintf(stderr,
      "\nRedundant call to comm_wait_for_client, ignored\n");
    return(0);
  }
```

```

  /* else comm_mode[channel] == HALFWAY_C which is correct */

  if ((sd[channel]=accept(ld[channel],0,0))<0) {
    perror("INET Domain Accept");
    return(5);
  }

    /* set to non-blocking if specified, else default is blocking */
  if (control==NONBLOCK_COMM)
    fcntl(sd[channel],F_SETFL,O_NDELAY);

  comm_mode[channel] = READY_C;  /* success */

  return(0);  /* success */
}



int init_comm_server(channel,port_num,control)
int channel,port_num,control;
{
// BEGIN SCOUT THESIS CHANGE
  static int num_socs = 0; /* Number of sockets already established. */
  int rc;
  int addrlen;
  struct sockaddr_in name;
  struct sockaddr_in *ptr;
  struct sockaddr addr;
  struct hostent *hp, *gethostbyaddr();
  int err;

  // If you are the SERVER ROBOT, set up next available channel
  // for the new client robot
  if (port_num == PORT_ARIEL){
     channel = ++num_socs;
     port_num = BASE_CLIENT_PORT + num_socs + 1;
  }
// END SCOUT THESIS CHANGE

  /* create a "listen" socket to receive service requests */
  if ((ld[channel]=socket(AF_INET,SOCK_STREAM,6))<0) {
    perror("INET Domain Socket");
    return(1);
  }

  /* initialize fields in an Internet address structure */
  name.sin_family = (short int) AF_INET;
  name.sin_port = htons(port_num);
  name.sin_addr.s_addr = INADDR_ANY;

  /* bind the Internet address to the Internet socket */
  if (bind(ld[channel],(struct sockaddr *)&name,sizeof(name))<0) {
    close(ld[channel]);
    perror("INET Domain Bind");
    return(2);
  }

```

```
  /* find out the port number assigned to our socket */
  addrlen = sizeof(addr);
  if ((rc=getsockname(ld[ channel] ,&addr,&addrlen))<0) {
    perror("INET Domain getsockname");
    return(3);
  }

  /* now "advertise" the port number assigned to us */
  ptr = (struct sockaddr_in *) &addr;
/*  printf("\n\tSocket has port number: %d\n",ntohs(ptr->sin_port));  */

  /* mark socket as a passive "listen" socket */
  if (listen(ld[ channel] ,5)<0) {
    perror("INET Domain Listen");
    return(4);
  }

  /* wait for a client to contact us... (blocking) */
  comm_mode[ channel] = HALFWAY_C;
  if ((err=comm_wait_for_client(channel,control)) != 0) {
    return(err);
  }

  /* find out who's calling us */
/*if ((rc=getpeername(sd[ channel] ,&addr,&addrlen))<0) {
    perror("INET Domain getpeername");
    return(6);
  }  */

    /* "announce" the caller */
/*printf("\n\tCalling socket from host %s\n",inet_ntoa(ptr->sin_addr));
  printf("\n\t    has port number %d\n",ptr->sin_port);
  if ((hp=gethostbyaddr(&ptr->sin_addr,4, AF_INET)) != NULL) {
    printf("\tFrom hostname: %s\n\tWith aliases:",hp->h_name);
    while (*hp->h_aliases)
      printf("\n\t\t\t%s",*hp->h_aliases++);
    printf("\n\n");
  }
  else {
    perror("\n\tgethostbyaddr() failed");
    printf("\n\th_errno is %d\n\n",h_errno);
  }  */

  comm_mode[ channel] = READY_C;
  return(0);
}



int init_comm_client(channel,mach_name,port_num,control)
int channel;
char *mach_name;
int port_num,control;
{
  struct sockaddr_in name;
  struct hostent *hp, *gethostbyaddr();

  /* create a "client" socket to request service */
```

```
  if ((sd[channel]=socket(AF_INET,SOCK_STREAM,0))<0) {
    perror("INET Domain Socket");
    return(1);
  }

  /* initialize fields in an Internet address structure */
  name.sin_family = AF_INET;
  name.sin_port = htons(port_num);
  hp=gethostbyname(mach_name);
  memcpy(&name.sin_addr.s_addr,hp->h_addr,hp->h_length);

  if (connect(sd[channel],(struct sockaddr *)&name,sizeof(name))<0) {
    perror("Connect()");
    return(2);
  }

    /* set to non-blocking if specified, else default is blocking */
  if (control==NONBLOCK_COMM)
    fcntl(sd[channel],F_SETFL,O_NDELAY);

  comm_mode[channel] = READY_C;  /* success */
  return(0);
}



int read_comm(channel,buf,bufsize)
int channel;
char *buf;
int bufsize;
{
  int nbytes;

// BEGIN SCOUT CHANGE
  // If no socket has been established on this channel,
  // then return.
  if (sd[channel] == 0){
     return(0);
  }
// END SCOUNT CHANGE

  if (comm_mode[channel] == READY_C) {
    memset(buf,0,bufsize);
    if ((nbytes=read(sd[channel],buf,bufsize))<0) {
      if (errno!=EWOULDBLOCK) {
      perror("INET domain Read");
      return(-1);  /* indicate error */
      }
      else    /* it was just an unblocked read with no data ready */
      return(0);
    }
    else if (nbytes==0) {
      fprintf(stderr,"\nSender Disappeared.\n");
      comm_mode[channel] = HALFWAY_C;
      return(-2);  /* no data to be read */
    }
    else {
      return(nbytes);  /* read data */
```

```
      }
    }
    else   /* socket has not been initialized */
      return(-3);
}


void write_comm(channel,buf,bufsize)
int channel;
char *buf;
int bufsize;
{
  if (comm_mode[channel] == READY_C)
    write(sd[channel],buf,bufsize);
}



demoserver()    /* demo */
{
  int a=0;
  int b=10;
  int c=100;
  char buf[256];

  init_comm_server(0,65003,NONBLOCK_COMM);
  while (1) {
    if (read_comm(0,buf,sizeof(buf))>0) {   /* if read something */
      sprintf(buf,"%d %d %d\0",a++,b++,c++);
      write_comm(0,buf,sizeof(buf));
    }
    sleep(1);
  }
}


democlient()
{
  int a,b,c;
  char buf[256];

  init_comm_client(0,"coyote",65003,BLOCK_COMM);
  while (1) {
    strcpy(buf,"Request");
    write_comm(0,buf,sizeof(buf));
    read_comm(0,buf,sizeof(buf));
    sscanf(buf,"%d %d %d",&a,&b,&c);
    printf("\nreceived %d %d %d\n",a,b,c);
    fflush(stdout);
    sleep(1);
  }
}
```

LIST OF REFERENCES

1. General Krulak's comments to the AFCEA/U.S. Naval Institute West '98 Conference, "The Race Goes to the Swiftest In Commercial, Military Frays," *Signal*, March 1998.

2. Gage, D. W., "Cost-Optimization of Many-Robot Systems," *Proceedings of SPIE Mobile Robots IX*, Boston, MA, November 1994.

3. Alptekin, G., "Geometric Formation with Uniform Distribution and Movement In Formation of Distributed Mobile Robots," Master's Thesis, Naval Postgraduate School, June 1996.

4. *NOMAD 200 Hardware Manual*, Nomadic Technologies, Inc., Mountain View, CA, 1997.

5. *Sensus 300: Infrared Ranging System*, Nomadic Technologies, Inc., Mountain View, CA, 1997.

6. *Sensus 200: Sonar Ranging System*, Nomadic Technologies, Inc., Mountain View, CA, 1997.

7. *Sensus 500: Laser Ranging System*, Nomadic Technologies, Inc., Mountain View, CA, 1997.

8. Latt, K., "Sonar-based Localization of Mobile Robots using the Hough Transform," Master's Thesis, Naval Postgraduate School, March 1997.

9. Glennon, J., "Feature-Based Localization of an Autonomous Mobile Robot using the Hough Transform and an Unsupervised Learning Network," Master's Thesis, Naval Postgraduate School, June 1998.

10. *SCOUT Beta 1.1*, Nomadic Technologies, Inc., Mountain View, CA, 1998.

11. *NOMAD SCOUT*, Nomadic Technologies, Inc., Mountain View, CA, 1997.

12. *Nomadic Host Development Environment*, Nomadic Technologies, Inc., Mountain View, CA, 1997.

13. *Mercury-RF1-TCP User's Guide Version 1.7*, Nomadic Technologies, Inc., Mountain View, CA, 1997.

14. *RangeLAN2 Access Point Models 7510 and 7520 User's Guide*, Proxim, Inc., Mountain View, CA, 1997.

15. Bornstein, J., Everett, H. R., and Feng, L., *Navigating Mobile Robots: Systems and Techniques*, A K Peters, 1996.

16. Moravec, H. P. and Elfes, A., "High Resolution Maps from Wide Angle Sonar," *IEEE International Conference on Robotics and Automation*, St. Louis, MO, March 1985.

17. Martin, M. C. and Moravec, H. P., *Robot Evidence Grids*, CMU-RI-TR-96-06, Carnegie Mellon University, Pittsburgh, PA, March 1996.

18. Elfes, A., "Sonar-Based Real-World Mapping and Navigation," *IEEE Journal of Robotics and Automation*, Vol. RA-3, No. 3, pp. 249-265, 1987.

19. Moravec, H. P., "Sensor Fusion in Certainty Grids for Mobile Robots," *AI Magazine*, Vol. 9, No. 2, pp. 61-74, 1988.

20. Graves, K., Adams, W. and Schultz, A., "Continuous Localization in Changing Environments," *Proceedings of the 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation*, Monterey, CA, July 1997.

21. Moravec, H. P., "Certainty Grids for Mobile Robots," *Proceedings of the Workshop on Space Telerobotics*, Jet Propulsion Laboratory, Pasadena, CA, July 1987.

22. Yamauchi, B., Schultz, A. and Adams, W., "Mobile Robot Exploration and Map-Building with Continuous Localization," *Proceedings of the 1998 IEEE International Conference on Robotics and Automation*, Leuven, Belgium, May 1998.

23. Yamauchi, B., "A Frontier-Based Approach for Autonomous Exploration," *Proceedings of the 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation*, Monterey, CA, July 1997.

24. Klaus, B. and Horn, P., *Robot Vision*, The MIT Press, 1986.

25. Yamauchi, B., "Frontier-Based Exploration Using Multiple Robots," *Proceedings of the Second International Conference on Autonomous Agents*, Minneapolis, MN, May 1998.

26. Yun, X., EC 4300 Class Notes, Naval Postgraduate School, 1997.

27. MacKenzie, P. and Dudek, G., "Precise Positioning using Model-Based Maps," *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, San Diego, CA, May 1994.

28. Dudek, G., Jenkin, M., Milios, E. and Wilkes, D., *Reflections on Modelling a Sonar Range Sensor*, CIM-92-9, McGill University, Montréal, Québec, Canada, May 1996.

29. Gage, D. W., "Telerobotic Requirements for Sensing, Navigation, and Communications," *Proceedings of the 1994 IEEE National Telesystems Conference*, San Diego, CA, May 1994.

30. Yuta, S. and Premvuti, S., "Coordinating Autonomous and Centralized Decision Making to Achieve Cooperative Behaviors between Multiple Mobile Robots," *Proceedings of the 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Raleigh, NC, July 1992.

31. Gage, D. W., "Development and Command-Control Tools for Many Robot Systems," *Proceedings of SPIE Microrobotics and Micromechanical Systems*, Philadelphia, PA, April 1995.

32. Gage, D. W., "How to Communicate with Zillions of Robots," *Proceedings of SPIE Mobile Robots VIII*, Boston, MA, September 1993.

33. Gage, D. W., "Network Protocols for Mobile Robot Systems," *Proceedings of SPIE Mobile Robots XII*, Pittsburgh, PA, October 1997.

34. Yamauchi, B., "Mobile Robot Localization in Dynamic Environments Using Dead Reckoning and Evidence Grids," *Proceedings of the 1996 IEEE International Conference on Robotics and Automation*, Minneapolis, MN, April 1996.

35. Rekleitis, I., Dudek, G. and Milios, E., "Multi-Robot Exploration of an Unknown Environment, Efficiently Reducing the Odometry Error," *Proceedings of the International Joint Conference in Artificial Intelligence*, Nagoya, Japan, August 1997.

36. Singh, K. and Fujimura, K., "Map Making by Cooperating Mobile Robots," *Proceedings of the 1993 IEEE International Conference on Robotics and Automation*, Atlanta, GA, May 1993.

37. Gage, D. W., "Sensor Abstractions to Support Many-Robot Systems," *Proceedings of SPIE Mobile Robots VII*, Boston, MA, November 1992.

38. Jacobus, C. J., Mitchell, B. T., Jacobus, H. N., Watts, R. C., Taylor, M. J. and Salazar, A., "Man-Portable Command, Communication, and Control Systems for the Next Generation of Unmanned Field Systems," *Proceedings of SPIE Mobile Robots VII*, Boston, MA, November 1992.

39. Mays, E. J. and Reid, F. A., "Shepherd Rotary Vehicle: Multivariate Motion Control and Planning," Master's Thesis, Naval Postgraduate School, September 1997.

40. Leonardy, T., "Implementation and Evaluation of an INS System for the Shepherd Rotary Vehicle," Master's Thesis, Naval Postgraduate School, December 1997.

41. Wang, J., Premvuti, S. and Tabbara, A., "A Wireless Medium Access Protocol (CSMA/CD-W) for Mobile Robot based Distributed Robotics Systems," *Proceedings of the 1995 IEEE International Conference on Robotics and Automation*, Nagoya, Japan, May 1995.

42. Everett, H. R., Gage, D. W., Gilbreath, G. A., Laird, R. T. and Smurlo, R. P. "Real-World Issues in Warehouse Navigation," *Proceedings of SPIE Mobile Robots IX*, Boston, MA, November 1994.

BIBLIOGRAPHY

Durrant-Whyte, H. F., *Integration, Coordination and Control of Multi-Sensor Robot Systems*, Kluwer Academic Publishers, 1988.

Stevens, W. R., *TCP/IP Illustrated, Volume 3*, Addison-Wesley, 1996.

Wright, G. R., Stevens, W. R., *TCP/IP Illustrated, Volume 2*, Addison-Wesley, 1995.

INITIAL DISTRIBUTION LIST

No. Copies

1. Defense Technical Information Center..2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library..2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Director, Training and Education 1
   MCCDC, Code C46
   1019 Elliot Rd.
   Quantico, VA 22134-5027

4. Director, Marine Corps Research Center..2
   MCCDC, Code C40RC
   2040 Broadway Street
   Quantico, VA 22134-5107

5. Director, Studies and Analysis Division...1
   MCCDC, Code C45
   300 Russell Road
   Quantico, VA 22134-5130

6. Marine Corps Representative..1
   Naval Postgraduate School
   Code 037, Bldg. 234, HA-220
   699 Dyer Road
   Monterey, CA 93940

7. Marine Corps Tactical Systems Support Activity.......................................1
   Technical Advisory Branch
   Attn: Maj J.C. Cummiskey
   Camp Pendleton, CA 92055-5080

8. Douglas W. Gage...1
   SPAWARSYSCEN D371
   53406 Woodward Road
   San Diego, CA 92152-7383

9. John A. Roese (PL-TS) 1
SPAWARSYSCEN D103
53570 Silvergate Ave Rm 3027A
San Diego, CA 92152-5271

10. Alan C. Schultz 1
Navy Center for Applied Research in Artificial Intelligence
Code 5514
Naval Research Laboratory
Washington, DC 20375-5337

11. Chairman, Code EC 1
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, CA 93943-5121

12. Professor Xiaoping Yun, Code EC/YX 2
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, CA 93943-5121

13. Professor Harold Titus, Code EC/TS 2
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, CA 93943-5121

14. Captain Patrick A. Hillmeyer, USMC 3
8769 Cold Plain Court
Springfield, VA 22153-2424